Instytut Podstawowych Problemów Techniki
Polska Akademia Nauk

Odwzorowanie procedur całkowania numerycznego w metodzie elementów skończonych na architektury procesorów masowo wielordzeniowych

Filip Krużel

Promotor: dr hab. inż. Krzysztof Banaś, prof. AGH

Instytut Informatyki
Wydział Fizyki, Matematyki i Informatyki
Politechnika Krakowska

Kraków, 2017

Podziękowania

Pragnę wyrazić serdeczną wdzięczność mojemu promotorowi, dr hab. inż. Krzysztofowi Banasiowi, za opiekę merytoryczną oraz umożliwienie rozwoju naukowego.

Szczególnie podziękowania składam mojej żonie Beatce, za cierpliwość i wsparcie w czasie pisania niniejszej pracy.

Pracę tą dedykuję pamięci mojego dziadka Czesława Pajdaka, którego ciągły pęd do nauki i nowoczesnych zdobyczy techniki jest dla mnie wzorem.

Spis treści

Rozdział 1

Wprowadzenie

1.1. Wstęp

W ostatnich latach nastąpił gwałtowny postęp w dziedzinie wykorzystywania architektur sprzętowych posiadających budowę masowo-wielordzeniową oraz heterogeniczną, do celów obliczeń naukowo-technicznych. Związane jest to z niemożliwością dalszego zmniejszania tranzystorów i innych układów elektronicznych z powodu zjawisk kwantowych oraz ograniczeń technologii [70]. Wymusiło to poszukiwanie innych metod tworzenia sprzętu i co za tym idzie zmiany w dotychczas istniejących algorytmach oraz metodach programowania. Dwa główne trendy w projektowaniu nowoczesnych architektur polegają z jednej strony na zwiększaniu liczby rdzeni zaangażowanych w przeprowadzanie obliczeń, a z drugiej na wyposażaniu tych rdzeni w specjalistyczne jednostki wspomagające w postaci coraz szerszych rejestrów wektorowych oraz jednostek arytmetyczno-logicznych na nich operujących. Pojawienie się tego typu układów nastręcza pytanie czy istniejące dotychczas algorytmy nadają się do adaptacji na nie, oraz równocześnie, jaka architektura byłaby dla nich najodpowiedniejsza. W niniejszej pracy autor postanowił znaleźć odpowiedź na to pytanie w odniesieniu do wybranych procedur metody elementów skończonych (MES) oraz zaproponować rozwiązania zmierzające do optymalnego ich projektowania na nowoczesne architektury, korzystające z wielopoziomowej hierarchii pamięci, jednostek wektorowych oraz masowej wielordzeniowości.

1.2. Stan badań

1.2.1. Rozwój architektur komputerowych

W związku z rozwojem nowoczesnych architektur komputerowych wystąpiła potrzeba przystosowania istniejących, oraz opracowania nowych, algorytmów korzystających ze wszystkich możliwości sprzętu. W dziedzinie obliczeń naukowo-technicznych konieczność zwiększenia dokładności symulacji numerycznych wiąże się ze wzro-

stem ilości niezbędnych operacji, a co za tym idzie, z koniecznością posiadania sprzętu o ogromnej mocy. Potrzeba skalowalności obliczeń zaowocowała rozwojem architektur równoległych i rozproszonych, w których wiele jednostek obliczeniowych może równocześnie pracować nad rozwiązaniem tego samego problemu.

Jednym z trendów w dziedzinie informatyki obliczeniowej jest wykorzystanie klastrów, specjalistycznych komputerów złożonych z wielu niezależnych jednostek obliczeniowych połączonych szybkimi sieciami komunikacyjnymi. Obliczenia na klastrach charakteryzują się wysokim stopniem skomplikowania i potrzebą korzystania z mechanizmów pamięci rozproszonej w celu synchronizacji obliczeń. Ich głównymi zaletami jest możliwość skalowalności w zależności od potrzeb oraz funduszy, ale ich minusem są ogromne koszty oraz skomplikowana warstwa programistyczna. Podstawą wydajności obliczeń na klastrach jest z jednej strony skalowalność, czyli utrzymanie optymalnego czasu obliczeń przy rosnących rozmiarach zadań oraz rosnącej liczbie jednostek w klastrze, a z drugiej wydajność na pojedynczym węźle klastra, w szczególności wykorzystanie możliwości mikroprocesorów i układów pamięci. To właśnie w tej ostatniej dziedzinie nastąpił największy rozwój i postęp w ostatnich latach. Dlatego autor w niniejszej pracy skupił się na architekturach pojedynczych mikroprocesorów wchodzących zazwyczaj w skład nowoczesnych klastrów obliczeniowych. Badania będące tematem niniejszej pracy mogą zostać bezpośrednio wykorzystane także dla klastrów, zwłaszcza w przypadku rozwiązań takich jak stosowany w pracy szkielet programistyczny ModFEM, w którym zagadnienia zrównoleglenia w modelu przesyłania komunikatów i wielowątkowości są w pełni odseparowane [87, 88].

W dziedzinie architektur mikroprocesorów używanych do celów obliczeniowych, w ostatnich latach wyodrębniły się dwie ścieżki rozwoju.

Pierwsza z nich bazuje na zwiększaniu liczby dostępnych rdzeni, co w przypadku architektur takich jak GPU, polega na wykorzystaniu rdzeni relatywnie prostych, z małą ilością dostępnych zasobów. Złożona budowa tego typu architektur nastręcza problemy związane z odpowiednim wykorzystaniem ich możliwości oraz wydajnym przesyłem danych pomiędzy różnymi poziomami pamięci dostępnymi dla programistów: rejestrami, pamięcią wspólną oraz pamięcią globalną [94, 2].

Kolejną ścieżką rozwoju nowoczesnych architektur mikroprocesorów jest zwiększanie szerokości rejestrów wektorowych w pojedynczym rdzeniu. Trend ten widoczny jest w nowoczesnych procesorach [50] oraz koprocesorach Intel Xeon Phi wyposażonych w szerokie 512 bitowe rejestry [54]. W przypadku koprocesorów liczba rdzeni jest zdecydowanie mniejsza niż w przypadku akceleratorów graficznych, a ich archi-

tektura jest kompromisem pomiędzy uniwersalnością CPU, a wyspecjalizowaniem GPU.

Na styku dwóch wymienionych ścieżek możemy wyodrębnić kilka architektur heterogenicznych, które próbują w jednej kości skupiać zarówno wyspecjalizowane jednostki o szerokich rejestrach (bazujące na budowie rdzeni GPU) oraz standardowe rdzenie ogólnego przeznaczenia. IBM Cell Broadband Engine, będąca architekturą w której rdzenie ogólnego przeznaczenia zostały połączone z wyspecjalizowanymi rdzeniami wektorowymi (Synergistic Processing Elements), stała się wzorem dla współczesnych architektur heterogenicznych, takich jak Intel Xeon Phi lub AMD Accelerated Processing Unit. Nowatorska architektura Cell/BE była podstawą do wielu badań z dziedziny obliczeń inżynierskich takich jak [60] lub [52].

1.2.2. Odwzorowanie obliczeń MES na architekturach procesorów

Pierwsze próby korzystania z niestandardowych sposobów wspomagania najbardziej wymagających części obliczeń MES, dotyczyły programowania kart graficznych (GPU). Pierwszą cytowaną pracą korzystającą z GPU była publikacja Wen Wu oraz Pheng Ann Heng [124], w której autorzy dokonali przeniesienia algorytmu mnożenia macierzy rzadkiej z solvera MES na kartę graficzną. Innymi przykładami wykorzystania kart graficznych dla wspomagania obliczeń MES było modelowanie trzęsień ziemi [64] lub implementacja różnych wariantów nieciągłej aproksymacji Galerkina [61].

Obliczenia metody elementów skończonych można podzielić na następujące etapy:

1. Dyskretyzacja przestrzeni na elementy skończone,
2. Całkowanie numeryczne w każdym elemencie,
3. Agregacja wyników do globalnej macierzy układu równań (macierzy sztywności),
4. Rozwiązanie dużego układu równań linowych.

W pewnych szczególnych przypadkach, jak na przykład solverach frontalnych [23, 51, 99] lub podejściach bezmacierzowych (matrix-free) [10, 55, 3] etapy 2, 3 i 4 stanowią całość i wymagają opracowania osobnych algorytmów. Jednakże w większości przypadków obliczenia są prowadzone za pomocą standardowego podejścia przedstawionego wcześniej.

Uwaga badaczy była dotychczas skupiona głównie na próbach optymalizacji finalnego rozwiązania układu równań, jako najbardziej czasochłonnej części obliczeń MES. Badania te prowadzono zarówno na standardowych procesorach CPU [123],

procesorach PowerXCell [122, 71], kartach graficznych [30, 33, 9], jak i koprocesorach Xeon Phi [109]. Prace badawcze poświęcone temu zagadnieniu, składają się głównie z prób zoptymalizowania procedury mnożenia macierz-wektor, która została szczegółowo przeanalizowana na różne architektury przez zespół prof. Wyrzykowskiego [125, 105] i innych badaczy [121, 103]. Oczywiście ze względu na, zazwyczaj, najdłuższy czas obliczeń, powyższe podejście jest jak najbardziej uzasadnione. Jednakże należy zwrócić uwagę, iż po optymalnym przyśpieszeniu tej procedury, wcześniejsze kroki obliczeń metody elementów skończonych także zaczynają znacząco wpływać na czas wykonania [79].

Z pozostałych prac, niektóre skupiają się na zagadnieniach całkowania traktowanego łącznie z asemblingiem [62, 39], gdzie lokalne macierze sztywności oraz wektory prawej strony mogą być od razu umieszczone w swoich globalnych odpowiednikach, tworząc główny układ równań liniowych do rozwiązania. Szczegółowe badania zaprezentowane w [61] dotyczą trójwymiarowego problemu nieciągłej aproksymacji Galerkina dla problemów hiperbolicznych. Charakterystyczne dla tego typu problemów jest podejście łączące fazę całkowania numerycznego z asemblingiem [58]. Pomimo tego, jak zauważono w [25], przy odpowiedniej optymalizacji trzeciego kroku MES, faza całkowania może zająć do 87% pozostałego czasu obliczeń. Jest to szczególnie widoczne w przypadku złożonych, nieliniowych sformułowań słabych oraz stosowania wyższych rzędów aproksymacji [19, 86, 120, 111].

Jedną z głównych wad stosowania asemblingu globalnej macierzy sztywności na akceleratorach jest konieczność wykorzystania dużej ilości pamięci do przechowywania macierzy. W związku z tym, optymalnym wydaje się stosowanie właściwie sformułowanych metod bezmacierzowych (matrix-free) [108, 3, 104, 78, 11]. Głównym problemem tego typu podejścia, jest konieczność wielokrotnej kalkulacji poszczególnych elementów macierzy w trakcie iteracyjnego rozwiązywania zadania. Wymusza to konieczność stosowania bardzo szybkiego algorytmu całkowania numerycznego, co jest jednym z powodów, dla którego warto rozważać ten algorytm osobno. W podejściu tego typu, elementy uzyskane z całkowania numerycznego są bezpośrednio wykorzystywane do rozwiązania układu równań bez korzystania z asemblingu globalnej macierzy. Kolejnym powodem dla którego całkowanie numeryczne może być rozważane osobno od procedury asemblingu, jest możliwość stosowania specyficznych dla konkretnego solwera metod generacji i przechowywania globalnych macierzy rzadkich [100]. Dodatkowo, procedura asemblingu może zostać przeprowadzona na innej jednostce obliczeniowej, pozwalając na pełne wykorzystanie możliwości sprzętu – takie podejście dla problemów wielkiej skali zostało zaprezentowane

w [26]. Pozwala to na elastyczne dostosowanie ostatecznego projektu aplikacji MES do możliwości sprzętu oraz do specyfiki danego problemu.

Dzięki swojej budowie, algorytm całkowania numerycznego pozwala na przetestowanie kilkudziesięciu kombinacji użycia wszystkich parametrów charakteryzujących nowoczesny sprzęt komputerowy, takich jak wielopoziomowa hierarchia pamięci, masowa wielordzeniowość oraz możliwości korzystania z jednostek wektorowych. Daje to pełen pogląd na możliwość wykorzystania danego urządzenia dla obliczeń MES, oraz daje wskazówki do wyboru odpowiedniego narzędzia dla danego algorytmu.

W wielu przypadkach badania odwzorowania na architekturę sprzętu skupiają się na opracowywaniu dedykowanych algorytmów na określone architektury oraz próbach ich optymalizacji [31, 119]. Podejście takie niejednokrotnie okazywało się być rozwiązaniem mało uniwersalnym i powodowało kłopoty przy przenoszeniu kodu na inną architekturę [107]. Podejmowane próby kreowania formalnych specyfikacji dla przenoszenia obliczeń MES na konkretne architektury, opierają się głównie na tworzeniu wyspecjalizowanych kompilatorów i wymagają bardzo szczegółowej wiedzy o charakterze rozwiązywanego problemu, algorytmów oraz budowie konkretnego sprzętu [29].

Dogłębne badania procesu otrzymywania macierzy sztywności MES na GPU (ze szczególnym uwzględnieniem assemblingu) zostały przeprowadzone w [13, 12]. W artykułach tych autorzy zaproponowali kilka różnych strategii podziału obliczeń pomiędzy wątki. Przetestowano też wydajności dla problemów liniowych i elementów dwu i trójwymiarowych. Opracowane tam techniki mogą stanowić podstawę badań nad optymalizacją agregacji globalnej macierzy dla różnych typów i stopni aproksymacji.

1.2.3. Narzędzia programistyczne

Próby rozwiązania problemu przenośności kodu pomiędzy różnymi typami akceleratorów zaowocowały powstaniem narzędzi programistycznych [41] ułatwiających programowanie oraz działających na różnych typach architektur. W badaniach nad różnymi metodami aproksymacji i asemblingu macierzy MES, szczegółowo zaprezentowanymi w [80, 82], zostały sprawdzone również różnice pomiędzy wersjami CUDA i OpenCL dla kart graficznych oraz możliwości przenoszenia wersji OpenCL między CPU a GPU. Rozwiązania te, mimo swojej przenośności, mają często problemy z równoczesnym zachowaniem wysokiej wydajności [82]. Potrzeba przenoszenia algorytmu wraz z zachowaniem jego wydajności zaowocowała próbami stworzenia narzędzi dla automatycznej generacji kodów MES dla CPU [59, 75, 74] oraz dla GPU [81, 101, 102]. Na szczególną uwagę zasługuje tu kompilator COFFEE

[76, 77], który pozwala na zastosowanie szeregu optymalizacji, takich jak odwrócenie kolejności oraz rozwijanie pętli, dopasowanie lub przesunięcie danych i inne. Metody te pozwalają na generację zoptymalizowanego kodu dla każdej architektury, próbując uzyskać maksymalne wykorzystanie sprzętu i jak największą wydajność. Autorzy zauważają, że mimo pozornej przenośności szkieletu programistycznego OpenCL, koniecznym jest wygenerowanie osobnego kodu dla każdej architektury. Inne podejście zaprezentowano w [27], gdzie dla uzyskania optymalnego kodu dla testowanych architektur użyto sieci neuronowej. W tejże pracy autorzy dokonali parametryzacji kilku benchmarków napisanych przy użyciu frameworka programistycznego OpenCL, a następnie użyli sieci neuronowej do znalezienia najbardziej optymalnej implementacji dla wszystkich badanych architektur. Badania zaprezentowane w tejże pracy pokazują, że dla każdej architektury inna kombinacja opcji prowadzi do osiągnięcia maksymalnej wydajności, a poprzez auto-tuning możliwym jest uzyskanie wysokiego stopnia przenośności przy minimalnych spadkach wydajności.

Dodatkowym problemem rozważanym przez badaczy jest problem zachowania odpowiedniej precyzji obliczeń. Karty graficzne i akceleratory starszej generacji nie pozwalały na dokonywane obliczeń podwójnej precyzji, a część nawet tych nowszych wykonuje te operacje kilka razy wolniej [91]. W wielu zastosowaniach MES stanowi to istotny problem związany z kumulacją błędów zaokrągleń, szczególnie przy stosowaniu bezpośrednich solwerów liniowych. Rozważanie całkowania numerycznego osobno od procedury rozwiązywania globalnego układu równań pozwala w takich wypadkach na zastosowanie metod mieszanej precyzji, w których rozwiązanie uzyskane z całkowania w pojedynczej precyzji może zostać poprawione w następnej fazie obliczeń [32, 4]. Istnieją również algorytmy bezpośrednie dla problemów zależnych od czasu, które nie wymagając rozwiązania globalnego układu równań, pozwalają na stosowanie rozwiązania w pojedynczej precyzji [63]. W związku z przybliżonym charakterem obliczeń MES, dla wielu zagadnień, procedura całkowania numerycznego wykonana w pojedynczej precyzji, nie generuje błędów na tyle dużych by były porównywalne z błędami samej dyskretyzacji [13]. Jednakże dla niektórych problemów, np. takich w których stosowane są różne sposoby rozwiązania w zależności od stopnia aproksymacji, koniecznym jest stosowanie podwójnej precyzji dla całkowania numerycznego [20].

1.3. Motywacja i cel pracy

Głównym celem pracy jest odpowiedź na pytanie jak efektywnie realizować tworzenie macierzy sztywności w równoległych symulacjach metodą elementów skoń-

czonych z wykorzystaniem wielopoziomowej hierarchii pamięci i architektur masowo wielordzeniowych. Jako wzorcowa metoda została wybrana procedura całkowania numerycznego dla sekwencji elementów. Algorytm ten, poprzez fazę obliczeń transformacji jakobianowych, jest bardziej skomplikowany niż zwykłe mnożenie macierzy oraz rozwiązywanie układu równań liniowych obecne w solwerach liniowych (algorytmach rozwiązywania układu równań liniowych). Dodatkową zaletą algorytmu jest to że podobne procedury można znaleźć na innych etapach metody elementów skończonych, takich jak projekcja rozwiązania czy obliczanie błędu metodą Zienkiewicza-Zhu [127]. Mimo tego, faza całkowania numerycznego jest często pomijana w badaniach naukowych na rzecz analizy innych faz obliczeń MES. Konieczność uwzględnienia algorytmu całkowania numerycznego w fazie optymalizacji obliczeń podkreśla fakt, iż w niektórych przypadkach może on zająć ok. 80% całkowitego czasu obliczeń [24].

Przedstawiony w poprzednim podrozdziale przegląd dotychczasowych badań, wskazuje na istotny brak wszechstronnej analizy algorytmu całkowania numerycznego w MES na architektury współczesnych mikroprocesorów, uwzględniającej wszystkie istotne dla wydajności aspekty, takie jak masowa wielordzeniowość, hierarchia pamięci, występowanie rozbudowanych jednostek wektorowych czy heterogeniczna budowa mikroprocesorów.

Wszystkie te powody sprawiają, że autor uznał za istotne zajęcie się analizą algorytmu całkowania numerycznego, która dzięki kompleksowemu podejściu i różnym wariantom może się stać wzorem w jaki sposób należy realizować także inne algorytmy na architekturach takich jak akceleratory, procesory graficzne oraz procesory wielordzeniowe.

1.4. Zawartość pracy

1.4.1. Układ pracy

Praca ta zawiera pełną analizę badanego algorytmu oraz jego implementację na różne architektury masowo wielordzeniowe takie jak karty graficzne, koprocesor Intel Xeon Phi oraz procesor IBM PowerXCell 8i. Praca podzielona jest na rozdziały w których omawiane są zastosowane przez autora algorytmy, sprzęt używany w badaniach oraz prezentowane są wyniki oraz wnioski z badań. W następnym rozdziale autor opisuje badane architektury oraz analizuje ich wady i zalety. W rozdziale trzecim skupia się na ogólnym opisie całkowania numerycznego, badanych problemów oraz przedstawia algorytm i jego warianty. Rozdział czwarty zajmuje się opisem modeli programowania, używanych w pracy. Kolejny rozdział zawiera szczegółowy

opis algorytmów dla badanych zadań oraz opis architektur wraz z implementacją, testami i dyskusją wyników. Niniejszą rozprawę kończy rozdział podsumowujący uzyskane wyniki oraz wnioski.

1.4.2. Elementy nowatorskie

Poprzez szczegółową analizę różnych wariantów algorytmu całkowania numerycznego autor szukał odpowiedzi na pytanie w jaki sposób należy podchodzić do projektowania oraz implementacji procedur mających korzystać z różnorakich cech nowoczesnych procesorów takich jak masowa wielordzeniowość, wektorowość czy wielopoziomowa hierarchia pamięci. W tym celu opracowano nowatorski system automatycznego tuningu uruchamianego kodu, który dzięki parametryzacji jest w stanie dostosować się do specyficznych możliwości większości nowoczesnych architektur mikroprocesorów. Praca ta zawiera również szczegółowe analizy badanych architektur z uwzględnieniem rzadko występujących w literaturze koprocesorów numerycznych Intel Xeon Phi oraz szczegółowych aspektów budowy nowoczesnych procesorów ogólnego przeznaczenia (CPU).

Niniejsza praca jest rozwinięciem oraz konkluzją prac rozpoczętych od prób uzyskania wydajnej implementacji badanego algorytmu całkowania numerycznego na architekturę procesorów PowerXCell 8i. Opracowane badania i ich wyniki zostały zaprezentowane podczas międzynarodowych konferencji Parallel Processing and Applied Mathematics (Wrocław, 2009) oraz Higher Order Finite Element and Isogeometric Methods (Kraków, 2011). Efektem tych prac są także dwie publikacje w punktowanych czasopismach z dziedziny Informatyki [66, 67].

Następnie, autor opracował kolejne metody aproksymacji MES na potrzeby dalszych badań. Wiązało się to z rozwojem kodu ModFEM, służącego do wielorakich obliczeń metodą elementów skończonych [87]. Efektem tych prac było stworzenie modułu aproksymacji standardowej (2009), modułu liniowej teorii sprężystości (2010) oraz implementacja algorytmu obliczania błędu Zienkiewicza-Zhu [69]. Rozwój różnych metod aproksymacji MES pozwolił autorowi na dogłębne zapoznanie się z rozmaitymi algorytmami. Dodatkowo, umożliwił on przeprowadzanie analizy pod kątem ich implementacji na architektury masowo wielordzeniowe.

Kolejne prace skupiały się na analizie i implementacji wcześniej badanego algorytmu całkowania numerycznego na nowoczesne akceleratory graficzne Nvidia Tesla oraz koprocesor numeryczny Intel Xeon Phi. Rezultatem tych rozważań stały się kolejne publikacje prezentujące wyniki oraz opracowaną metodologię testowania i pomiaru wydajności dla różnych architektur [5, 65].

Efektem kolejnych badań jest analiza możliwości dokonywania obliczeń naukowo-technicznych na procesorach AMD Accelerated Processing Units, które jako pierwsze pozwalały na korzystanie z jednego obszaru pamięci przez rdzenie obliczeniowe o różnych architekturach (Heterogenous Unified Memory Architecture) [68]. Równocześnie autor dopracowywał opracowany przez siebie system automatycznego tuningu badanego algorytmu co pozwoliło na jego kompleksową analizę zaprezentowaną w [6].

Rozdział 2

Badane architektury

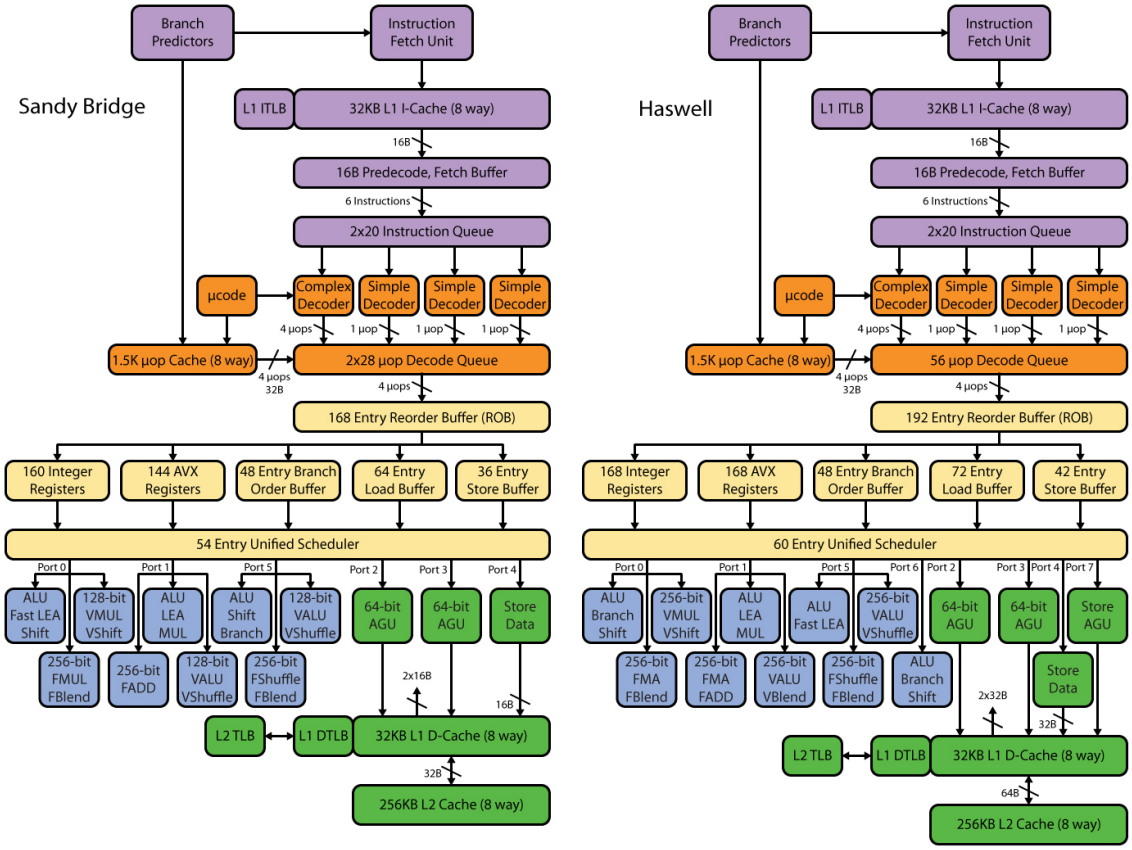
2.1. Nowoczesne procesory ogólnego przeznaczenia (CPU)

Rozwój współczesnych procesorów ogólnego przeznaczenia wiąże się z ciągłymi próbami powiększenia ich mocy poprzez różne zabiegi. Początkowe podnoszenie częstotliwości oraz zwiększanie ilości tranzystorów wraz ze zmniejszaniem ich wielkości, zostało uzupełnione przez stosowanie coraz większej liczby rdzeni obliczeniowych. Związane jest to z prawem Moore'a mówiącym, że liczba tranzystorów w układach scalonych podwaja się co 24 miesiące [89]. Obecny trend stosowania wielu rdzeni w procesie produkcji procesorów ogólnego przeznaczenia rozpoczęła firma IBM wprowadzając w 2001 roku procesor Power4 będący pierwszą jednostką posiadająca dwa rdzenie w jednym chipie [116]. Równocześnie już od lat osiemdziesiątych rozwijana była koncepcja wielowątkowości, szczegółowo opisana w [117]. Obie te koncepcje połączone zostały w architekturze "Niagara" procesora UltraSparc T1, gdzie czterowątkowe rdzenie zostały połączone w jeden układ CPU. Dla zastosowań domowych, pierwszym wielowątkowym procesorem był Pentium 4 zaprezentowany w 2002 roku i posiadający technologię Hyper-Threading, dzięki której praca mogła być dzielona pomiędzy dwa wirtualne wątki. Następnym krokiem w drodze do wielordzeniowości był procesor Intel Pentium Extreme Edition będący fizycznym sklejeniem dwóch rdzeni Pentium 4 w jednym chipie. Konstrukcja ta poprzez separację pamięci cache borykała się z dużymi problemami wydajnościowymi, co zostało rozwiązane w jej następcy, czyli architekturze Core 2. Równocześnie z konstrukcjami Intela, jego główny konkurent, firma Advanced Micro Devices (AMD) zaprezentowała swoje dwurdzeniowe procesory Athlon 64 X2.

Od czasu wprowadzenia pierwszych procesorów wielordzeniowych można zaobserwować ciągłe próby zwiększania ilości rdzeni, wraz ze zmniejszaniem wielkości tranzystorów je tworzących. Już w 2007 roku Intel zaprezentował prototypowy procesor Polaris posiadający 80 rdzeni, a w 2009 roku 48 rdzeniowy Single-Chip Cloud Computer. Rozwiązania sprzętowe zaprezentowane w tych procesorach, zostały

z sukcesem zaimplementowane w nowszych architekturach takich jak Sandy Bridge oraz Many Integrated Core (MIC).

Kolejnym krokiem mającym na celu zwiększenie możliwości obliczeniowych współczesnych procesorów ogólnego przeznaczenia, jest wyposażanie ich w specjalistyczne jednostki do operacji na wektorach. Dzięki temu, możliwym stało się szybsze wykonanie fragmentów programów odpowiadających paradygmatowi Single Instruction Multiple Data (SIMD), czyli takich, gdzie pojedyncza instrukcja jest stosowana równocześnie dla wielu danych. Pierwszą tego typu jednostką było, wprowadzone już w 1996 roku w procesorach Pentium, rozszerzenie MMX (MultiMedia eXtensions) pozwalające na operacje na 64 bitowych liczbach całkowitych. Kolejnymi jednostkami, rozszerzającymi możliwości wektorowe współczesnych procesorów, były 128 bitowe rejestry SSE, których wprowadzenie wiązało się z możliwością operacji na liczbach zmiennoprzecinkowych pojedynczej precyzji. Kolejne wersje rozszerzeń SSE wprowadziły obsługę liczb podwójnej precyzji oraz dodały dużą liczbę rozkazów wspomagających przetwarzanie wektorowe. Obecność tego typu rejestrów umożliwia również pakowanie niezależnych danych i ich przetwarzanie przy pomocy jednostek wektorowych.



Rysunek 2.1: Architektury Sandy Bridge oraz Haswell [57]

W ramach badań, autor dokonał analizy badanego algorytmu na procesorze opartym na architekturze Intel Sandy Bridge oraz jej następcy o nazwie kodowej Haswell. Intel Sandy Bridge była pierwszą architekturą CPU wyposażoną w 256 bitowe rejestry wektorowe AVX (Advanced Vector eXtension) pozwalające na przetwarzanie wektorów 4 liczb zmiennoprzecinkowych podwójnej precyzji lub 8 liczb o pojedynczej precyzji. Na rysunku 2.1 widać dokładnie budowę obu badanych procesorów z podziałem na poszczególne etapy przetwarzania potokowego.

Obszar fioletowy odpowiada za pobieranie instrukcji (Instruction Fetching). Jest on taki sam dla obu badanych procesorów i składa się z jednostki przewidywania rozgałęzień (Branch Predictors), jednostki pobierającej rozkazy (Instruction Fetch Unit), z której pobrane instrukcje umieszczane są w 32 kilobajtowej pamięci cache, wyposażonej dodatkowo w przyśpieszający tłumaczenie adresów wirtualnych bufor translacji (Translation Lookaside Buffer). Pobrane instrukcje są wstępnie dekodowane w 16 bajtowym buforze predekodowania, a następnie umieszczane w kolejce instrukcji podzielonej na dwie części (osobne dla każdego z wątków), z których każda jest w stanie przechowywać po 20 rozkazów.

Następna część schematu, oznaczona kolorem pomarańczowym, odpowiada za dekodowanie instrukcji (Instruction Decode). Składa się ona z 4 jednostek dekodujących instrukcje, które zamieniane są na ciąg mikrooperacji (μop) i umieszczane w zdekodowanej kolejce. Widać tutaj główną różnicę pomiędzy architekturami Sandy Bridge, gdzie cache ten jest podzielony sztywno na każdy wątek, a Haswell, gdzie całość cache jest dostępna dla obu wątków. Dodatkowo oba procesory wyposażone są w tak zwany cache mikrooperacji, który zachowuje się jak pamięć L0 instrukcji, dzięki której wstępnie zdekodowane instrukcje mogą być przetwarzane szybciej. Szczególnie widoczne jest to w przypadku stosowania instrukcji wektorowych bezpośrednio w kodzie - mogą one być zamienione na mikrooperacje już na etapie kompilacji i przetwarzane znacznie szybciej.

Następna część (żółta) schematu 2.1 odpowiada za planowanie wykonania poza kolejnością (Out-of-Order Scheduling). Polega to na zaplanowaniu wykonania w sposób jak najbardziej optymalny, a nie w kolejności wynikającej z organizacji kodu. W tym celu mikropoeracje są dzielone w buforze zmiany kolejności (Reorder Buffer), a następnie umieszczane w odpowiednich buforach (rejestry na dane, bufory operacji wczytywania i zapisu, bufor kolejności wykonania gałęzi). Dane z poszczególnych buforów są następnie odpowiednio kolejkowane przy pomocy schedulera (Unified Scheduler) i wysyłane do dalszego przetwarzania. Jak wydać na rysunku, główne różnice pomiędzy architekturami polegają na zwiększeniu pojemności poszczególnych buforów dla nowszej architektury.

Niebieska część rysunku odpowiada za wykonanie instrukcji (Instruction Execute) i składa się z jednostek arytmetyczno-logicznych odpowiadających za wykonywanie operacji na danych. Jak widać w procesorze Haswell nastąpiło zwiększenie liczby jednostek przetwarzających o jednostki obsługujące instrukcje AVX2 z których najważniejszą jest tzw. FMA (fused multiply add) (wzór (2.1)), która dostępna wcześniej była jedynie w specjalistycznych procesorach sygnałowych (DSP) oraz akceleratorach (PowerXCell 8i).

$$a = b + c * d \tag{2.1}$$

Stosowanie instrukcji FMA znacznie przyśpiesza algorytmy korzystające z mnożenia i dodawania w jednym kroku operacji oraz poprawia ich dokładność operując na niezaokrąglonych składowych. Dodatkowo w architekturze Haswell zrezygnowano z jednostek arytmetyczno-logicznych operujących na liczbach 128 bitowych zastępując je 256-cio bitowymi.

Zielona część rysunku 2.1 odpowiada za obsługę pamięci L1 i L2 i składa się z jednostek generacji adresu (Adress Generation Unit) dla zapisów i odczytów oraz jednostki zapisu danych (Store Data) do pamięci L1. Znaczącym ulepszeniem zastosowanym w architekturze Haswell, jest umieszczenie dodatkowej jednostki zarządzającej generacją adresów dla zapisów co pozwala na wykonanie 2 odczytów i jednego zapisu w trakcie każdego cyklu zegara, co dla architektury Sandy Bridge było możliwe tylko przy specjalnym dopasowaniu danych w pamięci (256 bit) [57]. Każdy z procesorów posiada pamięci L1 oraz L2 o takiej samej pojemności, wyposażone we własne bufory TLB wspomagające wyszukiwanie danych. Jak zauważono, na opisywanym schemacie widać, że z punktu widzenia mocy obliczeniowej, główne ulepszenia w nowszej architekturze Haswell, polegają na dodaniu zestawu instrukcji AVX2 rozszerzającego dotychczasowy, co wiąże się ze zwiększeniem ilości dostępnych rejestrów z 144 na 160 na rdzeń oraz dodaniu instrukcji FMA.

Tabela 2.1: Testowane procesory Intel Xeon [44]

	Xeon E5 2620 (Sandy Bridge)	**Xeon E5-2699 v.3 (Haswell)**
Częstotliwość taktowania	2 GHz	2,3 GHz
Liczba rdzeni	6	18
Liczba wątków	12	36
Technologia wykonania	32 nm	22 nm
Cache L2	2 MB	4,5 MB
Cache L3	15 MB	45 MB

Używana w badaniach konfiguracja uwzględniała zarówno wielordzeniowość, wielowątkowość oraz konfigurację wieloprocesorową (2 procesory na system). Charakterystyki badanego sprzętu zostały zaprezentowane w tabeli 2.1.

Architektura charakteryzująca oba przebadane procesory jest na czas pisania tej pracy najbardziej aktualna i nowoczesna. Z tego względu, autor uznał za stosowne użycie jej jako referencyjnej w swoich badaniach nad akceleratorami oraz sposobami przenoszenia algorytmów na nie.

2.2. Procesory graficzne (GPU)

Większość współczesnych akceleratorów obliczeń wywodzi się bezpośrednio z kart graficznych. W rozwoju specjalistycznych jednostek do przetwarzania grafiki można zauważyć kluczowy fakt połączenia wielu jednostek funkcjonalnych odpowiadających za różne etapy przetwarzania grafiki w jednym układzie GPU (Graphic Processing Unit). Ewolucja kart graficznych rozpoczęła się od koncepcji potoku (przetwarzania informacji na strumieniu danych), na której to bazowało wczesne przetwarzanie grafiki. Przetwarzanie danych graficznych w trzech wymiarach wymagało całego ciągu operacji - od generacji siatki po jej transformację w dwuwymiarowy obraz złożony z pikseli. Na przestrzeni lat, kolejne kroki tego przetwarzania były stopniowo przenoszone z układu CPU na wspomagającą kartę graficzną. Pierwszym układem GPU pozwalającym na przetwarzanie całego "potoku" operacji potrzebnych do wyświetlenia grafiki był układ Nvidia GeForce 256. Minusem tej konstrukcji było uniemożliwienie jakichkolwiek zmian w uprzednio przygotowanym strumieniu instrukcji, dlatego w 2001 roku Nvidia, w układzie GeForce 3, umożliwiła tworzenie specjalnych programów (shaderów) mogących operować na danych w potoku.
Jednostki wykonujące poszczególne operacje takie jak triangulacja, oświetlenie, transformacja węzłów siatki lub finalny rendering były implementowane jako fizycznie osobne fragmenty karty graficznej - zwane jednostkami cieniującymi. Kolejnym krokiem w ewolucji architektur GPU była unifikacja wszystkich jednostek cieniujących i stworzenie w pełni programowalnych multiprocesorów strumieniowych (Streaming Multiprocessor). Dzięki temu programowanie kart graficznych zbliżyło się możliwościami do programowania CPU, a w wielu zastosowaniach stało się wydajniejsze. Architektura G80 stała się podstawą pierwszego akceleratora obliczeń inżynierskich Nvidia Tesla na którego budowie opierają się wszystkie współczesne akceleratory GPGPU [18].

W ramach niniejszej rozprawy autor dokonał badań dwóch konkurencyjnych rozwiązań - jednego firmy Nvidia a drugiego AMD/ATI. Pierwszą badaną kartą

Rysunek 2.2: Tesla K20m [91]

jest akcelerator Nvidia Tesla K20m. Oparty jest on na architekturze rdzeni Kepler GK110 (Rys. 2.2). Jak widać na rysunku akcelerator ten posiada 13 Multiprocesorów nowej generacji SMX dzielących wspólny cache drugiego poziomu (L2). Za rozdział pracy pomiędzy poszczególne multiprocesory odpowiada scheduler nazwany Giga-Thread Engine. Akcelerator ten wyposażony jest w 6 osobnych kontrolerów pamięci dla obsługi 5GB pamięci RAM. Z jednostką macierzystą (host) łączy się za pomocą szybkiego interfejsu PCI Express 3.0 zdolnego do przepustowości teoretycznej na poziomie 16GB/s.

Każda z jednostek SMX jest wyposażona w 192 rdzenie przetwarzające pojedynczej precyzji oraz 64 rdzenie podwójnej precyzji, co łącznie daje 2496 rdzeni SP oraz 832 rdzenie DP (Rys. 2.3). Z powodu ogromnej ilości rdzeni konstrukcje takie określamy jako architektury masowo-wielordzeniowe. Każdy multiprocesor wyposażony jest w 32 jednostki wczytywania i zapisu danych (LD/ST), które przechowywane są w 65536 32-bitowych rejestrach (256 kB). Dodatkowo SMX posiada 32 jednostki specjalne (Special Function Unit) służące do wykonywania tzw. instrukcji transcendentalnych (sin, cos, odwrotność itp.). Wątki grupowane są w tzw. *warpy* (32 wątki), a ich pracą zarządzają 4 jednostki kolejkowania warpów (Warp scheduler). Każdy SMX wyposażony jest w 64 kB pamięci dzielonej pomiędzy cache L1 oraz pamięć

Rysunek 2.3: Multiprocesor strumieniowy SMX [91]

wspólną dla wątków. Ich rozmiar może być zarządzany przez programistę w zależności od potrzeb. Dodatkowo każdy multiprocesor posiada 48 kB pamięci stałej (read-only) oraz pamięć tekstur, która może być traktowana jako bardzo szybka pamięć cache tylko do odczytu.

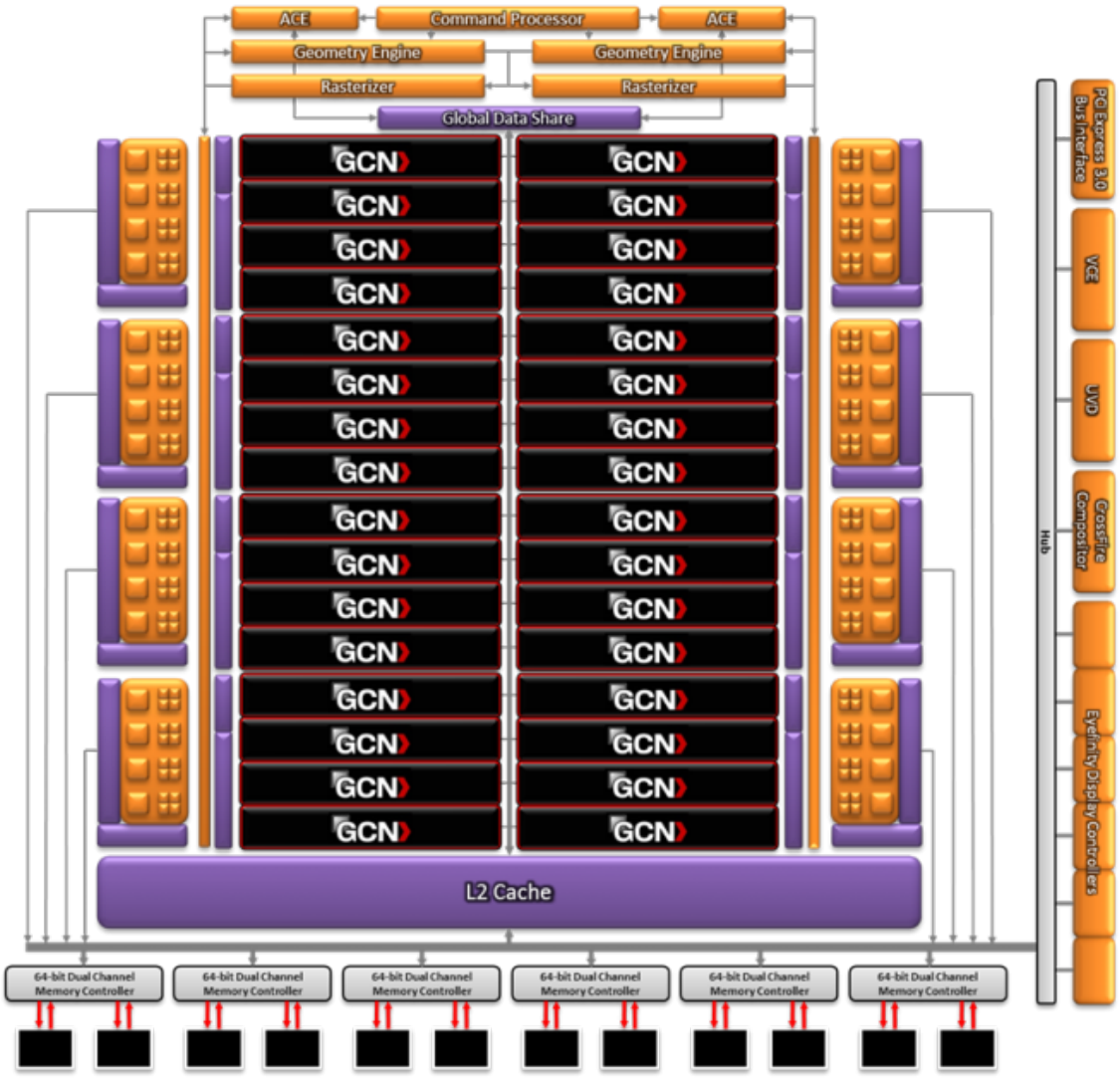
Parametry wydajnościowe badanego sprzętu przedstawia tabela 2.2.

Tabela 2.2: Parametry badanej karty Tesla K20m [91]

Teoretyczna wydajność obliczeń podwójnej precyzji [TFlops]	1.17
Teoretyczna wydajność obliczeń pojedynczej precyzji [TFlops]	3.52
Teoretyczna przepustowość pamięci [GB/s]	208

Teoretyczna wydajność dla badanej karty Tesla K20m jest liczona ze wzoru *2* (operacje w instrukcji FMA w każdym rdzeniu) ** liczba rdzeni * częstotliwość*

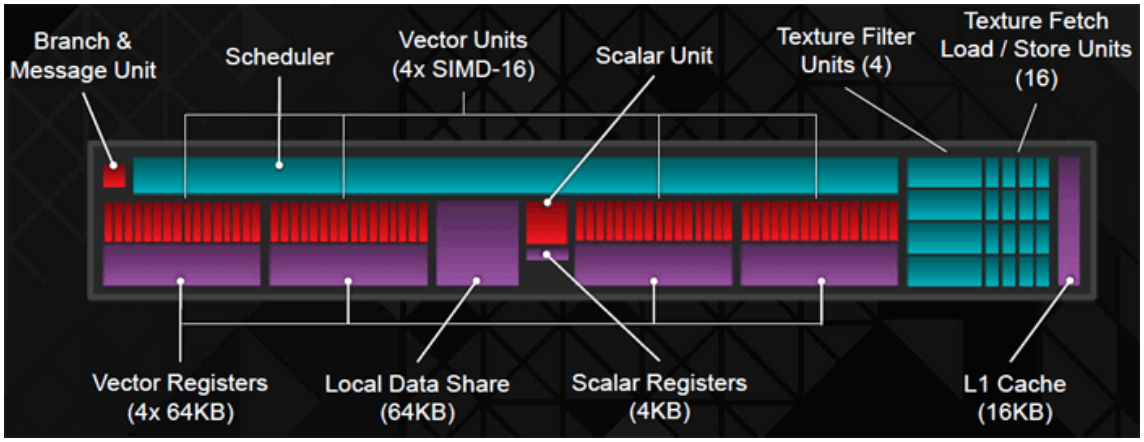
zegara (706GHz). Przepustowość pamięci została obliczona ze wzoru *2* (transfery danych w takcie - 2 dla pamięci typu Double Data Rate) * *szerokość magistrali* (320 bit) *2 (ilość kanałów pamięci) * *częstotliwość zegara* (1300MHz).



Rysunek 2.4: Radeon R9 280X [1]

Kolejną badaną architekturą była karta graficzna Gigabyte Radeon R9 280X oparta na architekturze o nazwie Tahiti XT (Rys. 2.4). Wyposażona jest ona w 32 rdzenie w mikroarchitekturze Graphics Core Next (GCN) oraz 3GB pamięci RAM. Posiada ona dwa asynchroniczne silniki obliczeń (ACE), dwie jednostki do obliczeń geometrii oraz rasteryzacji oraz pojedynczą jednostkę przetwarzania komend (Command Processor). Akcelerator posiada 8 jednostek renderujących (ROP) wyposażonych we własne bufory pamięci. Ponadto dane mogą być przechowywane w pamięci L2 oraz specjalnej 64 kB pamięci GDS (Global Data Share). Grupy złożone z czterech jednostek GCN posiadają współdzieloną 16 kB pamięć cache tylko do odczytu oraz

32 kB cache instrukcji. Do komunikacji z pamięcią RAM karty służy 6 osobnych dwukanałowych kontrolerów pamięci. Dodatkowo, na poziomie komunikacji z pamięcią systemu gospodarza, zastosowano interfejs PCI Express 3.0 oraz specjalne jednostki dekodowania wideo (Video Codec Engine oraz Unified Video Decoder). Karta jest również wyposażona w interfejs Crossfire pozwalający na połączenie ze sobą czterech kompatybilnych kart oraz w kontrolery technologii Eyefinity pozwalającej na wyświetlanie obrazu na sześciu monitorach równocześnie.



Rysunek 2.5: Multiprocesor strumieniowy GCN [91]

Jednostki GCN składają się z czterech 16 rdzeniowych jednostek wektorowych równoważnych rdzeniom w architekturze Nvidia Kepler (Rys. 2.5). Łącznie daje to 2048 rdzeni których wydajność zależy od tego czy obsługują liczby pojedynczej lub podwójnej precyzji oraz ustawień specyficznych dla danej architektury. Każda jednostka GCN zawiera scheduler odpowiadający za rozdział zadań pomiędzy wątki grupowane w fale (64 wątki). Każda jednostka wektorowa zawiera 64 kB plik rejestrów, a pojedyncza jednostka skalarna posiada 8 kB rejestrów. Pojedyncze jednostki GCN wyposażone są w 16 kB pamięci cache L1. Dodatkowo każdy GCN posiada 64 kB pamięci lokalnej (Local Data Share) służącej do synchronizacji wątków oraz przechowywania współdzielonych danych.

Parametry wydajnościowe badanej karty prezentuje tabela 2.3.

Tabela 2.3: Parametry badanej karty Radeon R9 280X [112]

Teoretyczna wydajność obliczeń podwójnej precyzji [TFlops]	0.87
Teoretyczna wydajność obliczeń pojedynczej precyzji [TFlops]	3.48
Teoretyczna przepustowość pamięci [GB/s]	288

Teoretyczna wydajność dla badanej karty Radeon R9 280X jest liczona z tego samego wzoru co w przypadku karty Tesla, czyli: *2* (operacje w instrukcji FMA

w każdym rdzeniu) ** liczba rdzeni * częstotliwość zegara* (850GHz). Dla badanego sprzętu wydajność obliczeń DP stanowi $\frac{1}{4}$ wydajności obliczeń SP. Przepustowość pamięci została obliczona ze wzoru *2* (transfery danych w takcie) ** szerokość magistrali* (384 bit) ** 2* (ilość kanałów pamięci) ** częstotliwość zegara* (1500MHz).

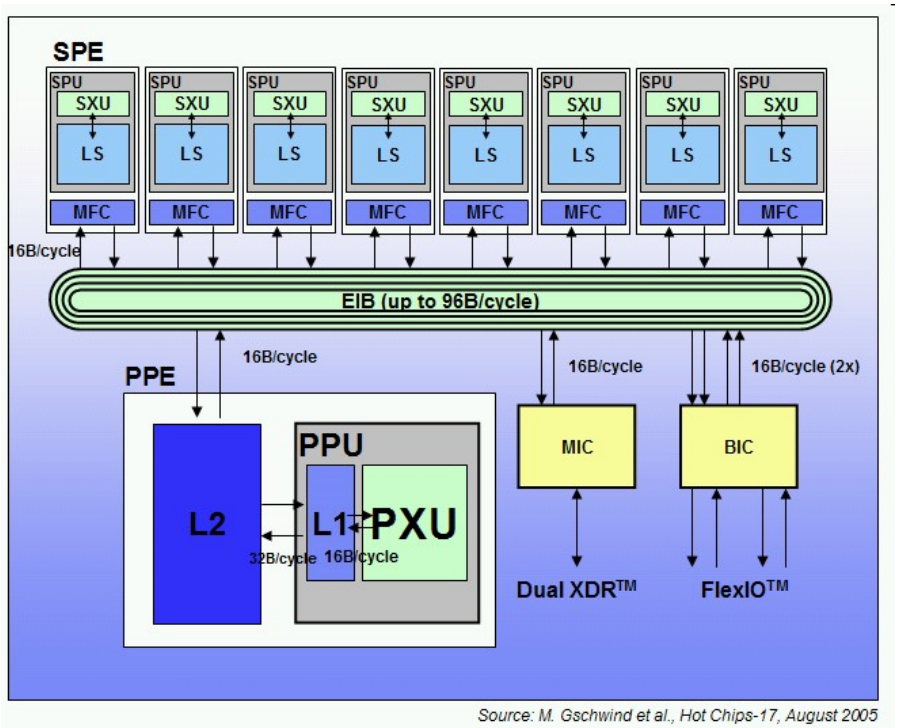
Jak widać w tabelach 2.2 oraz 2.3, obie karty charakteryzuje zbliżona wydajność, mimo tego, że rozwiązanie firmy Nvidia jest kartą dedykowaną do obliczeń wysokiej wydajności, a karta AMD jest konstrukcją przeznaczoną do grania w gry z zaawansowaną grafiką. Dzięki temu porównaniu można zauważyć pewne prawidłowości dotyczące budowy tego typu akceleratorów i wysnuć ciekawe wnioski dotyczące zarówno przewidywanej wydajności, jak i opłacalności zakupu dedykowanych rozwiązań, zwykle kilkadziesiąt razy droższych niż ich powszechnie dostępne odpowiedniki. Rozważania te wraz z wynikami badań są treścią kolejnych rozdziałów niniejszej pracy.

2.3. IBM PowerXCell 8i

W związku z rozwojem technologi GPGPU (General-Purpose computing on Graphics Processing Units) powstała idea stworzenia jednostki łączącej zalety szybkich wektorowych rdzeni GPU z rdzeniami ogólnego przeznaczenia obecnych w CPU. Trend ten, szczególnie obecny w dzisiejszych urządzeniach przenośnych (smartfonach, tabletach, laptopach) został zapoczątkowany w architekturze Cell Broadband Engine [43]. W ramach niniejszej pracy korzystano z procesora IBM PowerXCell 8i, będącego wariacją wyżej wymienionej architektury, rozszerzoną o jednostki przetwarzania podwójnej precyzji. Architektura ta składa się z jednego standardowego rdzenia PPE (Power Processor Element), zgodnego z architekturą x86, oraz ośmiu specjalistycznych rdzeni SPE (Synergistic Processor Elements) (Rys. 2.6).

Rdzenie SPE są wyspecjalizowanymi jednostkami z szerokimi rejestrami wektorowymi i osobną pamięcią prywatną. W przeciwieństwie do GPU, jednostki te nie charakteryzuje masowa wielordzeniowość, lecz dzięki większej ilości pamięci lokalnej są bardziej uniwersalne i mogą być używane przez szersze spektrum algorytmów.

Powstanie tego typu architektury było próbą przełamania wąskiego gardła z jakim spotykają się badacze korzystający z osobnych kart do akceleracji obliczeń, czyli wolnego połączenia pomiędzy akceleratorem a CPU oraz pamięcią RAM. Prędkość połączeń pomiędzy różnymi poziomami pamięci stała się głównym problemem dla architektur wspierających wymagające obliczenia [35]. Wymusza to projektowanie algorytmów z wysokim stopniem lokalności odniesień, oraz sprzętu będącego w stanie dostarczyć wystarczająco dużo pamięci jak najszybszego poziomu. W przeciwieństwie do dotychczasowych akceleratorów oraz CPU, pamięć lokalna procesora



Rysunek 2.6: Architektura Cell/BE

PowerXCell może być bezpośrednio zarządzana przez programistę, co wraz z jej szybkością może znacząco przyśpieszyć obliczenia.

Jednostka PPE procesora PowerXCell 8i zawiera w sobie 512 kB pamięci cache L2 oraz rdzeń PPU (Power Processor Unit) będący standardową jednostką IBM PowerPC z 32 kB pamięci L1. PPE potrafi przetwarzać dwa wątki na raz dzięki dwukierunkowej implementacji równoczesnej wielowątkowości (SMT - simultaneous multi-threading). Jednostki SPE składają się z kontrolera pamięci MFC (memory flow controller) oraz rdzenia SPU (Synergistic Processor Unit). SPU posiada 128 128-bitowych rejestrów, jednostki wykonawcze wykonujące operacje SIMD oraz 256 kB pamięci lokalnej. Dodatkowo, każda jednostka SPE wykonuje operacje na dwóch potokach, jednym dla liczb całkowitych lub zmiennoprzecinkowych, a drugim dla operacji odczytu/zapisu danych. Dla pojedynczego cyklu jest ona w stanie wykonać po jednej operacji dla każdego potoku - w szczególności dotyczy to operacji FMA (2.1) dla rejestrów SIMD. Jako że jedna operacja FMA odpowiada ośmiu standardowym operacjom arytmetycznym dla liczb pojedynczej precyzji, daje to wydajność 25,6 GFlops (przy referencyjnym taktowaniu rdzenia - 3,2 GHz). Równocześnie, drugi z potoków jest w stanie dokonać odczytu lub zapisu 16 bajtów danych z/do pamięci lokalnej każdego z rdzeni. Staranny wybór algorytmu w którym stosunek lokalnych odwołań do pamięci do operacji zmiennoprzecinkowych jest mniejszy

niż jeden, może doprowadzić do osiągnięcia wyników przybliżonych do teoretycznego maksimum. Zostało to pokazane w pracy [125] dla algorytmu mnożenia macierzy.

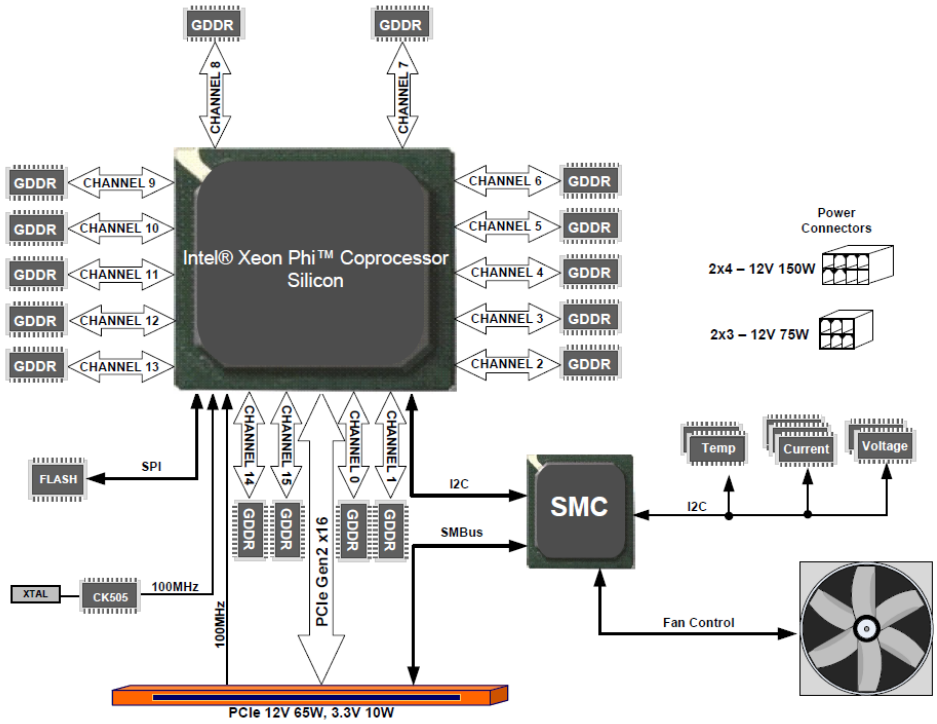
Każda część procesora PowerXCell 8i jest odpowiedzialna za różne zadania. Rdzeń Power odpowiedzialny jest za utrzymanie i zarządzanie zadaniami systemu operacyjnego oraz uruchamianie odpowiednich części programów na SPE. Wektorowe rdzenie SPE są przystosowane do jak najbardziej optymalnego przetwarzania zadanych fragmentów programu.

Jednostki SPE komunikują się z pozostałymi częściami procesora za pomocą bardzo szybkiego interfejsu EIB (element interconnect bus), który równocześnie pozwala wszystkim rdzeniom na komunikację z zewnętrzną pamięcią poprzez osobny interfejs MIC (memory interface controller). Zaprezentowane na rysunku 2.6 prędkości interfejsów połączone z 3,2 GHz częstotliwości taktowania zegara, pozwalają na osiągnięcie przepustowości 25.6 GB/s dla dostępu do pamięci lokalnych i globalnych oraz aż 307.2 GB/s dla interfejsu EIB. Ta ostatnia wartość, jednakże, zakłada trzy równoczesne transakcje dla każdego z czterech pierścieni EIB, co w praktyce jest trudne do uzyskania. Realnie, dla niektórych zastosowań, możliwym jest uzyskanie dwóch równoczesnych transakcji, co daje 204.8 GB/s teoretycznej przepustowości.

Mimo stosunkowo starej architektury, procesor PowerXCell można uznać za pierwowzór dla wszystkich nowoczesnych rozwiązań stosowanych zarówno w procesorach heterogenicznych (np. AMD APU), jak i nowoczesnych koprocesorach Intel Xeon Phi. Z tego względu, autor uznał za istotne przebadanie tej architektury pod względem użyteczności w obliczeniach naukowo-technicznych na przykładzie algorytmu całkowania numerycznego w MES.

2.4. Intel Xeon Phi

Wykorzystanie bardzo szerokich rejestrów wektorowych, charakteryzujące architekturę Cell/BE, było inspiracją dla firmy Intel przy tworzeniu architektury kart graficznych o nazwie Larabee. Równocześnie, firma ta próbowała pokonać główną barierę przeszkadzającą w większym rozpowszechnieniu technik programowania akceleratorów, którą był skomplikowany model i sposób programowania. Głównymi zaletami projektowanej architektury były bardzo szerokie (512-bitowe) rejestry wektorowe, specjalistyczne jednostki teksturujące, koherentna hierarchia pamięci oraz kompatybilność z architekturą x86 [110]. Równocześnie, Intel rozwijał swoje projekty Single Chip Computer oraz Teraflops Research Chip, które charakteryzowała bardzo duża wielordzeniowość. Na bazie tych projektów opracowana została architektura Intel MIC (Many Integrated Core), która znalazła zastosowanie w koprocesorach Intel



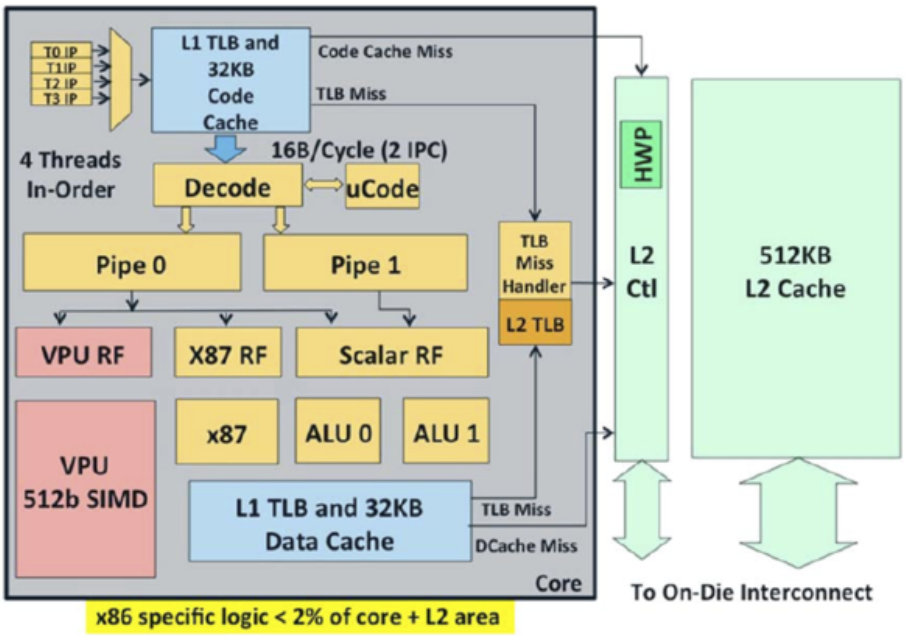
Rysunek 2.7: Schemat procesora Xeon Phi

Xeon Phi [34] o kodowej nazwie Knights Corner (KNC). Koprocesory te mają postać kart rozszerzeń połączonych z systemem macierzystym przy pomocy interfejsu PCI-Express (Rys. 2.7). Posiadają one aż 16 kanałów pamięci, oraz osobny kontroler zarządzania systemem (System Management Controller), który oprócz zarządzania przesyłem danych, steruje też pracą wentylatora (opcjonalnego) oraz monitoruje temperaturę karty. Akceleratory te wyposażone są we własny system operacyjny oparty na Linuxie, który ładowany jest na wewnętrzną pamięć flash. W zależności od wersji, dla programistów dostępne jest 57–61 rdzeni ze wsparciem sprzętowej wielowątkowości (4 wątki na rdzeń).

Tabela 2.4: Testowane procesory Intel Xeon Phi [44]

	5110P	**7120P**
Częstotliwość taktowania	1,05 GHz	1,24 GHz
Liczba rdzeni	60	61
Ilość pamięci RAM	8 GB	16 GB
Liczba wątków	240	244

Dla celów tej pracy wykorzystane zostały dwa koprocesory, 5110P oraz 7120P różniące się liczbą dostępnych rdzeni oraz ilością pamięci RAM. Charakterystyka wybranych koprocesorów została przedstawiona w tabeli 2.4. Ze względu na to, że ko-

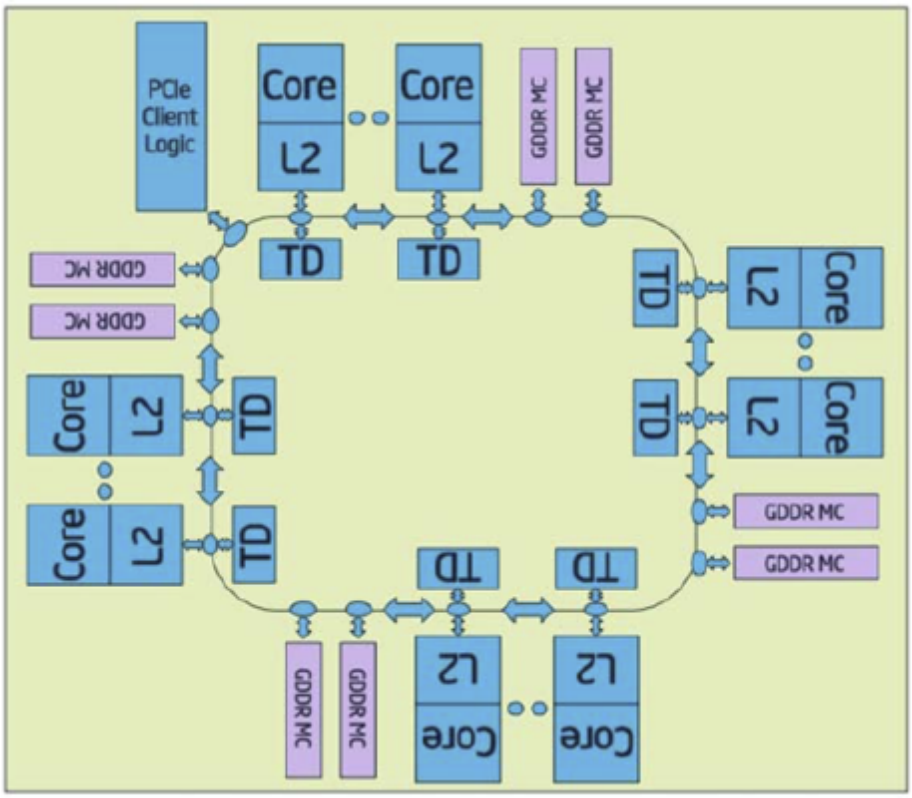


Rysunek 2.8: Pojedynczy rdzeń Xeon Phi

procesory te wyposażone są we własny system operacyjny dla którego rezerwowany jest jeden z rdzeni oraz fragment pamięci, dla programisty dostępne jest odpowiednio 236 i 240 wątków oraz 5,6 i 12 GB pamięci RAM [106].

Architektura rdzeni KNC bazuje głównie na architekturze Pentium 4, ale jest wzbogacona o 512-bitowe rejestry wektorowe. Kompatybilność z architekturą x86 ma teoretycznie pozwalać na łatwy transfer istniejących kodów w celu poprawy ich wydajności. Rysunek 2.8 przedstawia strukturę wewnętrzną pojedynczego rdzenia koprocesora. W budowie rdzenia można zauważyć sprzętowe wsparcie dla czterech wątków oraz dwa potoki przetwarzania. Rdzeń wyposażony jest ponadto w 32 kB pamięci cache L1 na dane oraz taką samą ilość pamięci na cache kodu. Dodatkowo posiada 512 kB pamięci cache drugiego poziomu (L2). Główną cechą rdzeni akceleratora Intel Xeon Phi jest wyspecjalizowana jednostka pozwalającą na przetwarzanie wektorowe (Vector Processing Unit), zaznaczona na schemacie kolorem czerwonym. Każdy VPU jest w stanie przetwarzać 8 liczb podwójnej precyzji lub 16 liczb pojedynczej precyzji oraz posiada wbudowane jednostki (Extended Math Unit) do obsługi operacji transcendentnych, podobnie jak w przypadku GPU. Rozszerzony zestaw instrukcji dla jednostki VPU został przez producenta nazwany IMCI i jest niestety niekompatybilny z instrukcjami AVX, co czyni problematycznym kwestie przenoszenia wcześniej skompilowanych kodów korzystających z wektoryzacji.

Każdy z rdzeni jest połączony bardzo szybkim interfejsem, co wraz z koherentną pamięcią cache, powoduje prawie natychmiastową wymianę danych pomiędzy nimi



Rysunek 2.9: Architekura Xeon Phi

(Rys. 2.9). Koherentność pamięci cache jest możliwa dzięki obecności tak zwanego katalogu znaczników (Tag Directory) dla każdego z rdzeni, połączonego bezpośrednio z pamięcią L2. Pamięć GDDR jest umieszczona pomiędzy rdzeniami co pozwala na bardziej równomierny rozkład pracy oraz sprzyja wydajnemu podziałowi danych pomiędzy rdzenie.

Omawiane koprocesory pozwalają na dokonywanie obliczeń na kilka sposobów. Jednym z nich jest tryb natywny dla którego cały program uruchamiany jest na akceleratorze. Drugim jest tryb offload, gdzie określone przez programistę fragmenty kodu oblicza koprocesor. Podobnym trybem charakteryzuje się użycie szkieletu programowego OpenCL, gdzie na koprocesorze uruchamiana jest część kodu w postaci osobnej procedury (tzw. kernela).

Koprocesory Intel Xeon Phi są kolejnym krokiem ku stworzeniu wysokowydajnej architektury do wspomagania obliczeń naukowo-technicznych. Z tego względu, autor uznał za zasadne objęcie badaniami tej architektury i sprawdzenie jej użyteczności w stosunku do badanego algorytmu całkowania numerycznego, a co za tym idzie jej ogólnej przydatności do celów obliczeń przy wykorzystaniu Metody Elementów Skończonych.

2.5. AMD APU

Przełomowa architektura procesora hybrydowego PowerXCell oprócz swoich zalet związanych z wydajnością, charakteryzowała się jeszcze jedną wyjątkową cechą. Była nią bardzo duża energooszczędność oraz stosunek Flops/Jul dzięki któremu w 2008 roku aż 7 pierwszych superkomputerów na liście Green500 stanowiły maszyniy oparte na architekturze Cell/BE [37]. Zainspirowało to producentów sprzętu do prób stworzenia bardzo wydajnych energooszczędnych procesorów, głównie do zastosowań w urządzeniach mobilnych takich jak laptopy lub tablety. Rosnąca wydajność tego typu urządzeń połączona jest z potrzebą minimalizacji zużywanej energii. Wynikiem tego trendu jest rozwój architektur, które w jednym procesorze zawierają kilka specjalistycznych rdzeni (multiprocesorów) oraz kilka rdzeni ogólnego przeznaczenia. Używając tego typu procesorów można oszczędzać energię w czasie działania normalnych rdzeni, równocześnie zachowując wysoką wydajność w trakcie korzystania z rdzeni specjalistycznych do bardziej skomplikowanych zadań.

Oprócz niewątpliwych zalet dotyczących efektywności energetycznej, heterogeniczne procesory wyposażone są najczęściej w najnowsze technologie pozwalające na szybką wymianę informacji między rdzeniami różnych typów. Oprócz wewnętrznych połączeń ważną kwestią jest także odpowiedni i szybki dostęp do różnych poziomów pamięci. Głównym problemem staje się tu pamięć RAM komputera. Wraz z wprowadzeniem architektur łączących w sobie jednostkę graficzną (GPU) z procesorem ogólnego przeznaczenia (CPU), pojawiło się kilka koncepcji dostępu do wymienionej wyżej pamięci. Powszechnie stosowany podział pamięci na dwie niezależne od siebie części, został zakwestionowany wraz z pojawieniem się architektury Cell/BE, która pozwalała na korzystanie z bezpośredniego dostępu do tych samych obszarów pamięci (Direct Memory Access) przez rdzenie SPU oraz PPU [40]. Korzystanie z DMA było jednakże okupione bardzo skomplikowanym modelem programowania oraz trudnościami z właściwym dopasowaniem danych w pamięci. Wraz z pojawieniem się możliwości korzystania ze szkieletu programistycznego OpenCL, model programistyczny uległ uproszczeniu, jednakże uniemożliwił korzystanie ze wszystkich zalet zunifikowanego dostępu do pamięci dla wszystkich rdzeni procesora PowerXCell.

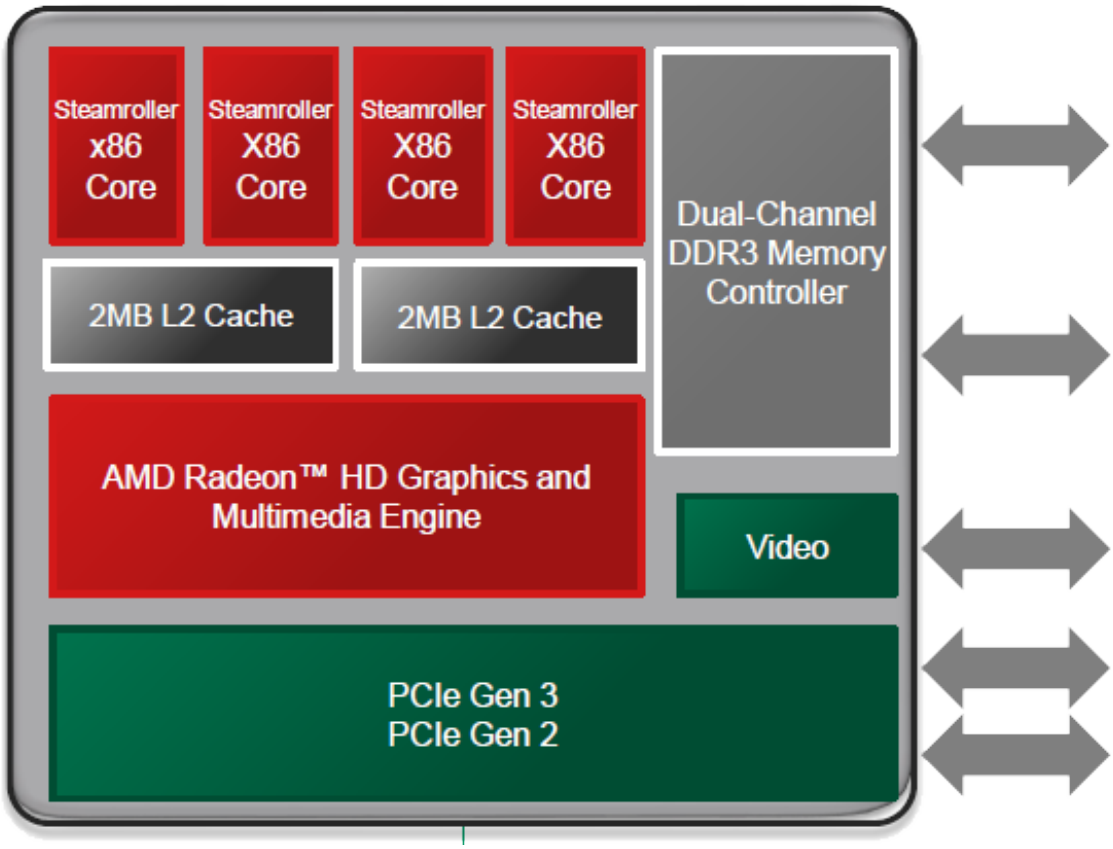
Problem wymiany danych pomiędzy różnymi typami rdzeni w ramach architektur heterogenicznych stanowi bardzo ważną kwestię. Z tego względu badacze pracują nad kilkoma rozwiązaniami, zarówno od strony sprzętowej jak i programowej. W czerwcu 2012 roku część z głównych producentów rozwiązań heterogenicznych, włączając AMD oraz ARM, założyło fundację Heterogeneous System Architecture (HSA). Fundacja ta, ma na celu opracowanie wspólnej techniki projektowania i pro-

gramowania architektur, zawierających w swojej strukturze jednostki obliczeniowe różnych typów i przeznaczenia [42]. Współpraca ta zaowocowała stworzeniem modelu jednolitego dostępu do pamięci dla architektur heterogenicznych (heterogeneous Unified Memory Model - hUMA), który znalazł zastosowanie w stworzonym przez AMD procesorze APU (Accelerated Processing Unit) [72].

Prawie równocześnie z opracowaniem modelu hUMA, konsorcjum Khronos Group odpowiadające za rozwój szkieletu programistycznego OpenCL zaprezentowało jego wersję 2.0. Pozwala ona na wykorzystanie możliwości sprzętowych modelu hUMA poprzez użycie współdzielonej pamięci wirtualnej (Shared Virtual Memory), dzięki której pamięć, dostępna dotychczas jako osobne obszary dla rdzeni GPU oraz CPU, została zunifikowana dla programisty [41]. W międzyczasie Intel opracował nową linię zintegrowanych jednostek graficznych o nazwie GT3, które mogą być opcjonalnie wyposażone we wbudowaną pamięć DRAM (embedded DRAM - eDRAM). Wraz z użyciem OpenCL 2.0 pamięć ta może być używana jako wspólna dla rdzeni CPU i GPU [36]. Firma Nvidia, wraz ze swoją architekturą GPU Maxwell oraz frameworkiem programistycznym CUDA 6.0 wprowadziła konkurencyjne rozwiązanie zunifikowanej pamięci wirtualnej (Unified Virtual Memory) mające być bezpośrednią odpowiedzią na model hUMA. Niestety rozwiązanie to oparte jest głównie na możliwościach programowych i zostało opracowane głównie w celu ułatwienia programistom tworzenia aplikacji korzystających z GPU [73]. Pomimo tego, w czasie konferencji Supercomputing 14 w listopadzie 2014 roku, Nvidia zaprezentowała nową architekturę GPU Pascal wyposażoną w nowy interfejs komunikacji pomiędzy GPU (jednym lub wieloma) a CPU nazwany NVlink [92].

Wszystkie te technologie wskazują na rozwój architektur heterogenicznych w kierunku wyeliminowania wąskiego gardła związanego z dostępem do danych dla różnego typu rdzeni obliczeniowych. Mimo tego, że główny nacisk w tworzeniu tych rozwiązań jest związany z przemysłem gier komputerowych, to opracowane technologie mogą być z powodzeniem wykorzystane w celach obliczeń naukowo-technicznych. W ramach niniejszej pracy, autor przetestował sztandarową jednostkę korzystającą z zalet sprzętowego modelu hUMA oraz programowego modelu SVM – AMD Accelerated Processing Unit A10-7850 o oznaczeniu kodowym 'Kaveri'.

AMD APU jest architekturą w której połączone zostały bardzo szybkie rdzenie SIMD obsługujące grafikę, z rdzeniami ogólnego przeznaczenia. Wyewoluowała ona z projektu Fusion, w którym firma AMD próbowała nowych sposobów komunikacji pomiędzy różnymi architekturami GPU oraz CPU. Rezultatem tych badań była, zaprezentowana w 2011 roku, pierwsza generacja procesorów Accelerated Processing Unit [118]. Wraz ze wspomnianym wcześniej założeniem fundacji HSA, AMD

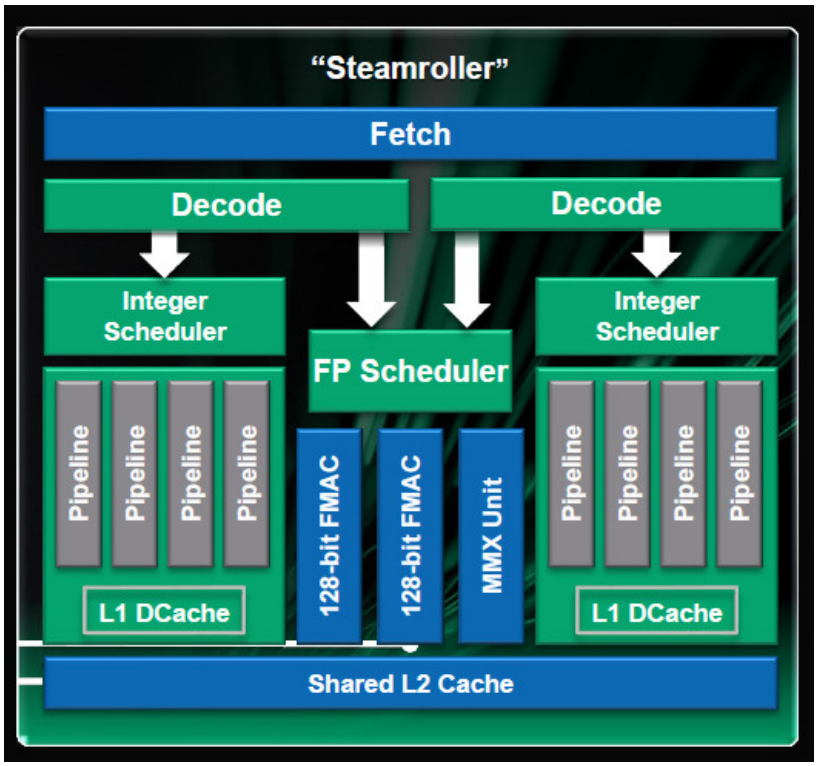


Rysunek 2.10: AMD APU A10-7850 [90]

APU ewoluowała w celu zgodności z głównymi założeniami paradygmatu architektur heterogenicznych. Wraz z trzecią generacją jednostek APU, nazwaną 'Kaveri', dostęp do pamięci dla rdzeni GPU oraz CPU został w pełni zintegrowany, pozwalając na przekazywanie danych pomiędzy CPU a GPU z użyciem całego dostępnego obszaru pamięci wirtualnej maszyny (heterogeneous Unified Memory Access). Jest to znaczące udogodnienie w stosunku do używanego dotychczas w stosunku do akceleratorów modelu NUMA (Non-Uniform Memory Access). Dodatkowo dzięki architekturze HSA, zostaje zachowana koherentność pamięci cache pomiędzy poszczególnymi rdzeniami co całkowicie eliminuje potrzebę operacji DMA podczas przemieszczania danych pomiędzy jednostką GPU a CPU [72]. Co więcej, wszystkie rdzenie mają takie same możliwości tworzenia i podziału pracy, w przeciwieństwie do standardowego modelu CPU+GPU, gdzie CPU przygotowuje i zarządza zadaniami do wykonania przez GPU. Ta technologia, nazwana heterogenicznym kolejkowaniem (heterogeneous Queuing), pozwala aplikacjom na generowanie kolejek zadań bezpośrednio na rdzeniach graficznych bez udziału rdzeni ogólnego przeznaczenia. Dodatkowo, GPU potrafi samo sobie generować zadania, dokładnie w taki sam sposób jak dotychczas robiło to CPU. Dzięki hQ, podczas wykonania programu można

znacząco zredukować potrzebę korzystania z systemu operacyjnego, co w istotny sposób minimalizuje opóźnienia i pozwala aplikacji na wykonywanie zadań na tych rdzeniach, które w danym momencie będą bardziej odpowiednie. Zalety architektury heterogenicznej wpływają również na sposób programowania, ułatwiając go i zbliżając do klasycznego pisania programów na CPU [38].

Badany procesor AMD Accelerated Processing Unit A10-7850 wyposażony jest w 12 rdzeni obliczeniowych - 4 rdzenie CPU oraz 8 GPU (Rys. 2.10). Posiada on wbudowany kontroler dwukanałowej pamięci typu DDR3 oraz kontrolery wyświetlania i komunikacji z modułem zarządzającym płyty głównej przez interfejs PCI-Express. Rdzenie CPU korzystają z architektury Steamroller, z bazową częstotliwością 3,7 GHz oraz 4 MB cache drugiego poziomu (Rys. 2.11).



Rysunek 2.11: Architektura AMD Steamroller [98]

Jest to architektura będąca bezpośrednim konkurentem opisywanej wcześniej, architektury Intel Haswell, dzięki czemu wspiera ona instrukcje FMA oraz operacje na 256 bitowych rejestrach AVX. Jak widać na rysunku, każdy dwurdzeniowy procesor Steamroller współdzieli scheduler oraz jednostki specjalistyczne dla operacji zmiennoprzecinkowych oraz 2 MB pamięci cache L2. Każdy z pojedynczych rdzeni procesora Steamroller wyposażony jest w 96 kB pamięci cache L1 na kod oraz 16 kB pamięci cache L1 na dane.

Rdzenie GPU jednostki APU, korzystają z opisywanej wcześniej architektury GCN (Rys. 2.5) i są równoważne z parametrami karty graficznej Radeon R7 o częstotliwościowi taktowania 720 MHz.

Architektura AMD APU, dzięki swojej przełomowej konstrukcji, jest doskonałym materiałem do badań nad nowoczesnymi sposobami wspierania obliczeń naukowych, m. in. w MES. Mimo tego, iż została stworzona głównie dla celów wykorzystania w wysokowydajnych laptopach, rozwiązania w niej zastosowane są odpowiedzią na wiele problemów nurtujących badaczy zajmujących się obliczeniami wysokiej wydajności. Dzięki równorzędnemu dostępowi do pamięci oraz możliwości równoważnego generowania zadań przez rdzenie różnych typów, spodziewany może być znaczący wzrost wydajności dla dotychczasowych, przy równoczesnym ułatwieniu tworzenia przyszłych programów. Badania autora mają na celu znalezienie odpowiedzi na pytanie, czy kierunek wyznaczony przez architekturę HSA, może być tym właściwym dla zastosowań w dziedzinie obliczeń naukowo-technicznych.

2.6. Podsumowanie

Zróżnicowanie dostępnych architektur do obliczeń wysokiej wydajności, implikuje konieczność znalezienia odpowiedniego sposobu na optymalne odwzorowanie posiadanych algorytmów na poszczególne procesory. Koniecznym wydaje się tu znalezienie właściwego sposobu projektowania lub adaptacji algorytmów, aby mogły korzystać w pełni z możliwości posiadanego sprzętu. Głównym problemem, wydaje się zachowanie wysokiej wydajności przy równoczesnej przenośności używanych algorytmów. Potrzeba całkowitego przeprojektowania algorytmu może być czynnikiem, który wpłynie na wybór danej architektury. Równocześnie dostosowując architekturę do konkretnego algorytmu, ograniczamy jej możliwości adaptacji i spektrum zastosowań. Z tego względu badania autora mają na celu opracowanie metod i wskazówek dotyczących projektowania zarówno algorytmów, jak i wyboru odpowiedniego sprzętu pod dane zastosowanie. Szczegółowe badania algorytmu całkowania numerycznego dają odpowiedź na pytanie, jak optymalnie odwzorowywać tego typu zagadnienia na nowoczesnych architekturach sprzętowych.

Rozdział 3

Całkowanie numeryczne

3.1. Definicja problemu

Metoda elementów skończonych służy do obliczenia przybliżonego rozwiązania równań różniczkowych cząstkowych, zdefiniowanych dla danego, współcześnie najczęściej trójwymiarowego, obszaru obliczeniowego Ω wraz z zadanymi warunkami na brzegu $\partial\Omega$ [126, 56, 8, 20]. W celu wykonania obliczeń zakłada się podział obszaru obliczeniowego na określoną liczbę elementów o prostej geometrii (czworościany, sześciany, elementy pryzmatyczne). Obliczenia wykonywane są korzystając z tzw. sformułowania słabego definiującego rozwiązywany problem. Ogólna postać sformułowania słabego dla problemów analizowanych w niniejszej pracy ma postać [56, 8]:

Znajdź niewiadomą funkcję $\boldsymbol{u}$, należącą do pewnej przestrzeni funkcji kawałkami wielomianowych, dla której równanie:

$$\begin{aligned}\int_\Omega \Bigg(\sum_i \sum_j \boldsymbol{C}^{i;j} \frac{\partial \boldsymbol{w}}{\partial \boldsymbol{x}_i} \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{x}_j} + \sum_i \boldsymbol{C}^{i;0} \frac{\partial \boldsymbol{w}}{\partial \boldsymbol{x}_i} \boldsymbol{u} + \\ + \sum_i \boldsymbol{C}^{0;i} \boldsymbol{w} \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{x}_i} + \boldsymbol{C}^{0;0} \boldsymbol{w} \boldsymbol{u} \Bigg) d\Omega + \mathrm{WBL} = \\ = \int_\Omega \left(\sum_i \boldsymbol{D}^i \frac{\partial \boldsymbol{w}}{\partial \boldsymbol{x}_i} + \boldsymbol{D}^0 \boldsymbol{w} \right) d\Omega + \mathrm{WBP},\end{aligned} \tag{3.1}$$

jest spełnione dla każdej funkcji testowej $\boldsymbol{w}$ zdefiniowanej w tej samej (lub lekko zmodyfikowanej) przestrzeni funkcji.

W powyższym wzorze $\boldsymbol{C}^{i;j}$ oraz $\boldsymbol{D}^i$, $i, j = 0, .., N_D$ oznaczają współczynniki zależne od rozwiązywanego problemu (N_D - ilość wymiarów przestrzeni), a WBP oraz WBL oznaczają odpowiednio wyrażenia z prawej i lewej strony związane z warunkami brzegowymi oraz całkami po brzegu obszaru $\partial\Omega$. W niniejszej pracy autor skupił się na obliczaniu całki po obszarze Ω ze względu na to, że całki brzegowe zazwyczaj są mniej wymagające obliczeniowo, a z punktu widzenia algorytmicznego powtarzają podobny schemat całkowania. Część obliczeń odpowiadającą za warunki

brzegowe w używanym przez autora kodzie, wykonują zawsze rdzenie CPU, przy pomocy standardowych metod MES.

Niewiadomą funkcję $\boldsymbol{u}$ oraz funkcję testującą $\boldsymbol{w}$ przedstawia się za pomocą kombinacji liniowych funkcji bazowych ψ^r $(r = 1, ..., N)$, których postać określa przestrzeń funkcji i zależy od wybranej metody aproksymacji. W swoich badaniach autor wykorzystywał metodę nieciągłej aproksymacji Galerkina oraz metodę standardowej aproksymacji linowej w zadaniach Poissona oraz konwekcji-dyfuzji. Pierwsza z tych metod definiuje funkcje bazowe jako rozszerzenie dla całego obszaru obliczeniowego elementowych funkcji kształtu ϕ^r $(r = 1, ..., N_S)$, będących lokalnymi wielomianami rzędu p, niezależnie dla każdego elementu. W metodzie liniowej aproksymacji standardowej, funkcje bazowe powstają przez odpowiednie "sklejenie" funkcji kształtu zdefiniowanych dla poszczególnych elementów. W pojedynczym elemencie funkcje kształtu mają postać funkcji liniowych lub wieloliniowych w zależności od używanego elementu bazowego (czworościenny lub pryzmatyczny). Bez względu na zastosowaną metodę aproksymacji, funkcje $\boldsymbol{u}$ oraz $\boldsymbol{w}$ przedstawiamy jako kombinację liniową funkcji bazowych w postaci:

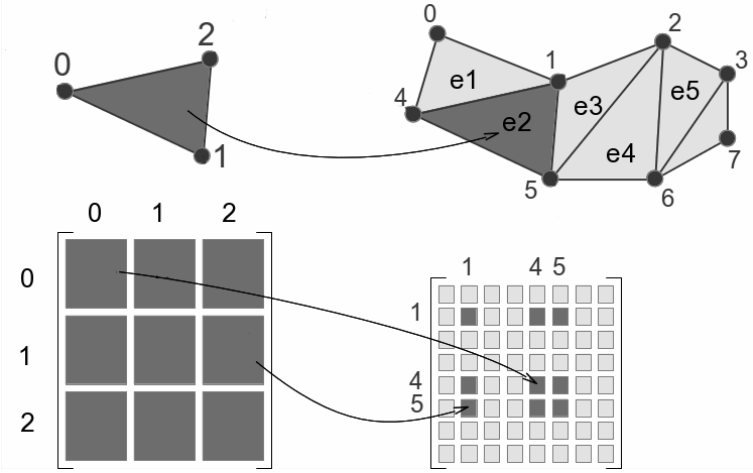
$$u = \sum_r U^r \psi^r, \qquad\qquad w = \sum_s W^s \psi^s \tag{3.2}$$

Wektor współczynników kombinacji liniowej funkcji bazowych $\boldsymbol{U}$ będący poszukiwanym rozwiązaniem numerycznym definiuje przybliżone rozwiązanie danego problemu MES. Jego rozmiar $\boldsymbol{N}$ jest głównym parametrem, określającym rozmiar rozwiązywanego zadania oraz związane z tym wymagania dotyczące zasobów obliczeniowych. Rozwiązanie zadania (3.1) może być jedynym krokiem do rozwiązania całego problemu, a może być też tylko jednym etapem bardziej złożonych obliczeń (np. problemu zależnego od czasu lub nieliniowego). Stanowi to kolejny powód dla którego algorytm całkowania numerycznego stanowi ważną część obliczeń MES wpływającą na ogólną wydajność symulacji.

Wektor $\boldsymbol{U}$ otrzymuje się jako rozwiązanie układu równań liniowych, którego macierz nazywana jest (globalną) macierzą sztywności. Pojedynczy wyraz globalnej macierzy sztywności jest całką po całym obszarze obliczeniowym, którą możemy przedstawić jako sumę całek po pojedynczych elementach:

$$\int_\Omega ...d\Omega = \sum_{e_i} \int_{\Omega_{e_i}} ...d\Omega \tag{3.3}$$

Każdy z obliczonych wyrazów odpowiada parze globalnych funkcji bazowych, co, ze względu na zerowanie się każdej z funkcji bazowych w całym obszarze obliczeniowym z wyjątkiem jednego lub kilku elementów, w większości przypadków daje



Rysunek 3.1: Schemat całkowania w MES

wynik zerowy. Powoduje to, że stworzona macierz sztywności jest macierzą rzadką. Liczba elementów niezerowych zależy od typu siatki (strukturalna lub nie), typu elementów oraz wybranej aproksymacji. Dla liniowej aproksymacji standardowej, liczba niezerowych elementów w wierszu macierzy jest rzędu kilkunastu. W przypadku nieciągłej aproksymacji Galerkina wyższych stopni p, liczba niezerowych elementów rośnie do kilkuset w wierszu. Niemniej jednak, w każdym przypadku, większość elementów macierzy równa się zero.

Całki obliczone dla funkcji kształtu pojedynczych elementów tworzą lokalne macierze sztywności dla każdego elementu. Wyrazy z tych macierzy mogą pojawiać się w kilku miejscach w globalnej macierzy sztywności (Rys. 3.1).

Podejście wybrane w niniejszej pracy bazuje na najczęściej stosowanym założeniu, że każdy element w obszarze obliczeniowym jest całkowany tylko jednokrotnie. Na skutek całkowania otrzymujemy małą elementową macierz sztywności, której pojedynczy wyraz obliczamy w postaci:

$$\begin{aligned}(A^e)^{rs} = \int_{\Omega_e} \Bigg(\sum_i \sum_j \boldsymbol{C}^{i;j} \frac{\partial \boldsymbol{\phi}^r}{\partial \boldsymbol{x}_i} \frac{\partial \boldsymbol{\phi}^s}{\partial \boldsymbol{x}_j} + \sum_i \boldsymbol{C}^{i;0} \frac{\partial \boldsymbol{\phi}^r}{\partial \boldsymbol{x}_i} \boldsymbol{\phi}^s + \\ + \sum_i \boldsymbol{C}^{0;i} \boldsymbol{\phi}^r \frac{\partial \boldsymbol{\phi}^s}{\partial \boldsymbol{x}_i} + \boldsymbol{C}^{0;0} \boldsymbol{\phi}^r \boldsymbol{\phi}^s \Bigg) d\Omega + \mathrm{WBL},\end{aligned} \tag{3.4}$$

gdzie r i s są lokalnymi indeksami zmieniającymi się w zakresie 1 do N_S – liczby funkcji kształtu dla danego elementu.

Analogicznie wektor prawej strony oblicza się ze wzoru:

$$(b^e)^r = \int_{\Omega_e} \left(\sum_i \boldsymbol{D}^i \frac{\partial \boldsymbol{\phi}^r}{\partial \boldsymbol{x}_i} + \boldsymbol{D}^0 \boldsymbol{\phi}^r \right) d\Omega + \mathrm{WBP}, \tag{3.5}$$

Podobnie jak poprzednio WBL i WBP oznaczają wyrazy związane z warunkami brzegowymi, tym razem zdefiniowane dla pojedynczego elementu i jego funkcji kształtu.

3.2. Algorytm całkowania numerycznego

W związku z tym, że elementowe funkcje kształtu w praktyce zdefiniowane są dla tzw. elementu referencyjnego danego typu i zastosowanej aproksymacji, istnieje potrzeba zastosowania odpowiedniej transformacji geometrycznej w siatce używanej do obliczeń. Każdy element siatki obliczeniowej traktowany jest jako odwzorowanie elementu referencyjnego za pomocą tejże transformacji. Oznaczając fizyczne współrzędne punktów w siatce jako $\boldsymbol{x}$, transformacja z elementu odniesienia o współrzędnych $\boldsymbol{\xi}$ ma postać $\boldsymbol{x}(\boldsymbol{\xi})$. Jest ona zazwyczaj uzyskiwana poprzez ogólną formę transformacji w postaci liniowych, wieloliniowych, kwadratowych, kubicznych itp. geometrycznych funkcji bazowych, oraz zestawu geometrycznych stopni swobody. Stopnie te są zazwyczaj związane z określonymi punktami w elemencie referencyjnym (w zależności od użytej aproksymacji) – np. wierzchołkami dla transformacji liniowej lub wieloliniowej.

Transformacja całkowania z elementu odniesienia wymaga użycia macierzy Jacobiego $J = \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\xi}}$. Obliczanie macierzy jakobianowej, jej wyznacznika oraz odwrotności, jest charakterystycznym elementem całkowania numerycznego w metodzie elementów skończonych i jak wspomniano w poprzednich rozdziałach, istotnie odróżnia ten algorytm od zwykłego całkowania oraz typowych algorytmów macierzowych. Sposób obliczania wyrazów macierzy jakobianowych różni się w zależności od stosowanego typu elementu oraz aproksymacji. Dla elementów geometrycznie linowych (np. czworościennych), wyrazy te są stałe, co znacząco zmienia sposób wykonywania algorytmu. Dla elementów wieloliniowych oraz aproksymacji wyższych rzędów, obliczenia tych wyrazów mogą stać się istotnym elementem wpływającym na wydajność algorytmu.

Zastosowanie zmiany zmiennych do elementu referencyjnego $\hat{\Omega}$ dla wybranej przykładowej całki, stosowanej w problemach rozwiązywanych w niniejszej pracy, prowadzi do wzoru:

$$\begin{aligned}\int_{\Omega_e} \sum_i \sum_j \boldsymbol{C}^{i;j} \frac{\partial \boldsymbol{\phi}^r}{\partial \boldsymbol{x}_i} \frac{\partial \boldsymbol{\phi}^s}{\partial \boldsymbol{x}_j} d\Omega = \\ = \int_{\hat{\Omega}} \sum_i \sum_j \boldsymbol{C}^{i;j} \sum_k \frac{\partial \hat{\boldsymbol{\phi}}^r}{\partial \boldsymbol{\xi}_k} \frac{\partial \boldsymbol{\xi}_k}{\partial \boldsymbol{x}_i} \sum_z \frac{\partial \hat{\boldsymbol{\phi}}^s}{\partial \boldsymbol{\xi}_z} \frac{\partial \boldsymbol{\xi}_z}{\partial \boldsymbol{x}_j} \det \boldsymbol{J} d\Omega,\end{aligned} \tag{3.6}$$

gdzie $\hat{\phi}^r$ oznacza funkcje kształtu dla elementu referencyjnego. We wzorze tym zastosowany jest wyznacznik macierzy Jacobiego $det\boldsymbol{J} = det(\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\xi}})$ oraz składowe macierzy jakobianowej transformacji odwrotnej $\boldsymbol{\xi}(\boldsymbol{x})$, z elementu rzeczywistego do elementu odniesienia.

Kolejnym krokiem jest zamiana całki analitycznej na kwadraturę numeryczną. W niniejszej pracy stosowana była kwadratura Gaussa ze współrzędnymi w elemencie odniesienia oznaczanymi jako $\boldsymbol{\xi}^Q$ i wagami $\boldsymbol{w}^Q$ gdzie $Q = 1, .., N_Q$ (N_Q - liczba zastosowanych punktów Gaussa, zależna od typu elementu oraz stopnia aproksymacji). Dla całki (3.6) prowadzi to do wzoru:

$$\int_{\Omega_e} \sum_i \sum_j \boldsymbol{C}^{i;j} \frac{\partial \boldsymbol{\phi}^r}{\partial \boldsymbol{x}_i} \frac{\partial \boldsymbol{\phi}^s}{\partial \boldsymbol{x}_j} d\Omega \approx \sum_{Q=1}^{N_Q} \left(\sum_i \sum_j \boldsymbol{C}^{i;j} \frac{\partial \boldsymbol{\phi}^r}{\partial \boldsymbol{x}_i} \frac{\partial \boldsymbol{\phi}^s}{\partial \boldsymbol{x}_j} \det \boldsymbol{J} \right) \Bigg|_{\boldsymbol{\xi}^Q} \boldsymbol{w}^Q \tag{3.7}$$

W związku z tym, że wszystkie części ogólnych wzorów (3.4) oraz (3.5) transformowane są analogicznie do całki (3.6), niniejsza praca analizuje jedynie powyższy wzór (3.7). Otrzymane wzory, zawierające odpowiednie sumy dla wszystkich współczynników C^{ij} i D^i, znajdują zastosowanie we wszystkich problemach analizowanych przez autora tj. stacjonarnym problemie Poissona oraz uogólnionym problemie konwekcji-dyfuzji-reakcji. Finalna postać wzoru (3.7) może zmieniać się w zależności od wybranej aproksymacji oraz typów elementów, ze względu na to, że niektóre wartości są stałe dla całego elementu. Dzieje się tak na przykład dla elementów geometrycznie liniowych jak np. czworościany. Z tego względu, procedura całkowania numerycznego musi zostać dokładnie przeanalizowana dla jak największej ilości tego typu przypadków, co będzie przeprowadzone w dalszej części tej pracy.

W celu zapisania ogólnego algorytmu całkowania numerycznego, wprowadzono pewne modyfikacje w notacji wzorów. Mają one na celu ujednolicenie oraz opis algorytmu od strony informatycznej. Są one następujące:

— $\boldsymbol{\xi}^Q[i_Q]$, $\boldsymbol{w}^Q[i_Q]$ – tablice z lokalnymi współrzędnymi punktów całkowania (punktów Gaussa) oraz przyporządkowanymi im wagami, $i_Q = 1, 2, ..., N_Q$, gdzie N_Q – liczba punktów Gaussa zależna od wybranej geometrii oraz typu i stopnia aproksymacji,

— $\boldsymbol{G}^e$ – tablica z danymi dotyczącymi geometrii elementu (związane z transformacją z elementu odniesienia do elementu rzeczywistego),
— $\mathbf{vol}^Q[i_Q]$ – tablica z elementami objętościowymi $\mathbf{vol}^Q[i_Q] = det\left(\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\xi}}\right) \times \boldsymbol{w}^Q[i_Q]$,
— $\boldsymbol{\phi}[i_Q][i_S][i_D]$, $\boldsymbol{\phi}[i_Q][j_S][j_D]$ – tablice z wartościami kolejnych lokalnych funkcji kształtu, oraz ich pochodnych względem globalnych ($\frac{\partial \phi^{i_S}}{\partial x_{i_D}}$) i lokalnych współrzędnych ($\frac{\partial \hat{\phi}^{i_S}}{\partial \xi_{i_D}}$) w kolejnych punktach całkowania i_Q,
  — $i_S, j_S = 1, 2, ..., N_S$, gdzie N_S – liczba funkcji kształtu zależna od wybranej geometrii oraz stopnia aproksymacji,
  — $i_D, j_D = 0, 1, ..., N_D$, gdzie N_D – wymiar przestrzeni. Dla i_D, j_D różnego od zera, tablice odnoszą się do pochodnych względem współrzędnej o indeksie i_D, a dla $i_D = 0$ do funkcji kształtu, stąd $i_D, j_D = 0, 1, 2, 3$,
— $\boldsymbol{C}[i_Q][i_D][j_D][i_E][j_E]$ – tablica z wartościami współczynników problemowych (dane materiałowe, wartości stopni swobody w poprzednich iteracjach nieliniowych i krokach czasowych itp.) w kolejnych punktach Gaussa, $i_E, j_E = 1, 2, ..., N_E$, gdzie N_E – liczba składowych wektorowych w rozwiązaniu - np. $N_E = 3$ dla liniowej teorii sprężystości gdzie współczynniki zależne są od tensora naprężeń o wymiarze 3x3,
— $\boldsymbol{D}[i_Q][i_D][i_E]$ – tablica z wartościami współczynników d^i w kolejnych punktach Gaussa,
— $\boldsymbol{A}^e[i_S][j_S][i_E][j_E]$ – tablica przechowująca lokalną, elementową macierz sztywności,
— $\boldsymbol{b}^e[i_S][i_E]$ – tablica przechowująca lokalny, elementowy wektor prawej strony.

Wprowadzając zaprezentowaną notację, można przedstawić ogólny wzór na elementową macierz sztywności:

$$\begin{aligned}\boldsymbol{A}^e[i_S][j_S][i_E][j_E] = \sum_{i_Q}^{N_Q} \sum_{i_D, j_D}^{N_D} \boldsymbol{C}[i_Q][i_D][j_D][i_E][j_E] \times \\ \times \boldsymbol{\phi}[i_Q][i_S][i_D] \times \boldsymbol{\phi}[i_Q][j_S][j_D] \times \mathbf{vol}^Q[i_Q],\end{aligned} \tag{3.8}$$

oraz wektor prawej strony:

$$\boldsymbol{b}^e[i_S][i_E] = \sum_{i_Q}^{N_Q} \sum_{i_D}^{N_D} \boldsymbol{D}[i_Q][i_D][i_E] \times \boldsymbol{\phi}[i_Q][i_S][i_D] \times \mathbf{vol}^Q[i_Q]. \tag{3.9}$$

Dzięki wprowadzonej notacji możliwym staje się stworzenie ogólnego algorytmu całkowania numerycznego dla elementów skończonych tego samego typu oraz stopnia aproksymacji (Alg. 1):

Algorytm 1: Uogólniony algorytm całkowania numerycznego dla elementów tego samego typu i stopnia aproksymacji

```
- określenie parametrów algorytmu – N_EL, N_Q, N_E, N_S;
- wczytaj tablice ξ^Q i w^Q z danymi całkowania numerycznego;
- wczytaj wartości wszystkich funkcji kształtu oraz ich pochodnych względem
  współrzędnych lokalnych we wszystkich punktach całkowania w elemencie odniesienia;
for e = 1 to N_EL do
    - wczytaj współczynniki problemowe wspólne dla wszystkich punktów całkowania
      (Tablica C^e);
    - wczytaj potrzebne dane dotyczące geometrii elementu (Tablica G^e);
    - zainicjuj elementową macierz sztywności A^e i elementowy wektor prawej strony b^e;
    for i_Q = 1 to N_Q do
        - oblicz potrzebne dane transformacji jakobianowych (∂x/∂ξ, ∂ξ/∂x, vol);
        - korzystając z macierzy jakobianowych oblicz pochodne funkcji kształtu
          względem współrzędnych globalnych;
        - oblicz współczynniki C[i_Q] i D[i_Q] w punkcie całkowania;
        for i_S = 1 to N_S do
            for j_S = 1 to N_S do
                for i_D = 0 to N_D do
                    for j_D = 0 to N_D do
                        for i_E = 1 to N_E do
                            for j_E = 1 to N_E do
                                A^e[i_S][j_S][i_E][j_E] += vol ×C[i_Q[[i_D][j_D][i_E][j_E]×
                                                          ×φ[i_Q][i_S][i_D] × φ[i_Q][j_S][j_D];
                            end
                        end
                        if i_S = j_S and i_D = j_D then
                            for i_E = 0 to N_E do
                                b^e[i_S][i_E] += vol ×D[i_Q][i_D][i_E] × φ[i_Q][i_S][i_D];
                            end
                        end
                    end
                end
            end
        end
    end
    - zapis całej macierzy A^e oraz wektora b^e;
end
```

Jak zauważono w poprzednim rozdziale, koszt wysłania danych na akcelerator i z powrotem, może być bardzo wysoki i powinien być ukryty poprzez odpowiednio duże nasycenie obliczeń. Sprzyja temu taki układ Algorytmu 1 w którym zewnętrzna pętla jest pętlą po wszystkich elementach obszaru obliczeniowego. Ogólna postać Algorytmu 1, w której nie uwzględniamy położenia danych w różnych poziomach pamięci, pozwala również na traktowanie każdej z wewnętrznych pętli jako niezależnych i zmianę ich kolejności w celu uzyskania optymalnej wydajności. Osiągnięte jest to także dzięki temu, iż każdą z potrzebnych danych możemy obliczyć wcześniej i używać w razie potrzeby. Pozwala to na głęboką analizę algorytmu i tworzenie jego wariantów w zależności od posiadanego sprzętu, co będzie celem dalszych rozważań.

3.3. Zadania testowe

3.3.1. Zagadnienie Poissona

Jednym z badanych zadań było proste równanie Poissona opisujące np. stacjonarny rozkład temperatury (3.10).

$$\nabla^2 u = f \tag{3.10}$$

Wynik dla tego typu zadania jest skalarny, w związku z czym liczba elementów wektora rozwiązania $N_E = 1$. Dla zadania Poissona macierze współczynników konwekcji-dyfuzji-reakcji $\boldsymbol{C}[i_Q]$ mają postać (3.11), dla wszystkich punktów całkowania.

$$\boldsymbol{C}[i_Q] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.11}$$

Z tego wynika że poszczególne wyrazy macierzy sztywności otrzymuje się uproszczonym wzorem 3.7 mającym postać:

$$(A^e)^{rs} = \int_{\Omega_e} \sum_i \frac{\partial \boldsymbol{\phi}^r}{\partial \boldsymbol{x}_i} \frac{\partial \boldsymbol{\phi}^s}{\partial \boldsymbol{x}_i} d\Omega \tag{3.12}$$

Wektor współczynników $\boldsymbol{D}[i_Q]$ ma postać:

$$\boldsymbol{D}[i_Q] = \begin{bmatrix} 0 & 0 & 0 & S_v \end{bmatrix} \tag{3.13}$$

gdzie S_v jest współczynnikiem całkowania prawej strony, różnym dla każdego punku całkowania.

3.3.2. Uogólniony problem konwekcji-dyfuzji-reakcji

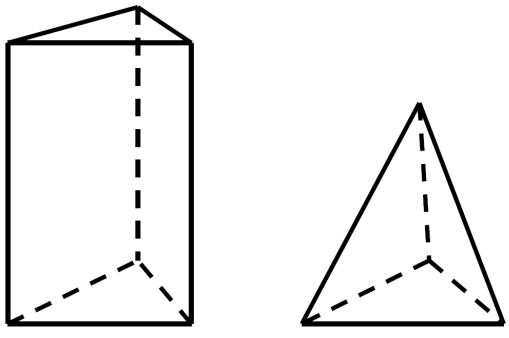
Kolejnym z badanych zadań było uogólnione zadanie konwekcji-dyfuzji-reakcji. Dla celów jak największego wykorzystania zasobów przyjęto założenie iż macierz $\boldsymbol{C}[i_Q]$ i wektor $\boldsymbol{D}[i_Q]$ współczynników równania będą w pełni wypełnione. Macierze tego typu pojawiają się np. w zadaniach konwekcyjnego transferu ciepła, po zastosowaniu stabilizacji SUPG.

W przypadku testowego zadania konwekcji-dyfuzji-reakcji, rozważane współczynniki $\boldsymbol{C}[i_Q]$ i $\boldsymbol{D}[i_Q]$ są stałe dla całego elementu. Ma to na celu uwolnienie od specyfiki konkretnych zastosowań, przy zachowaniu zwiększonej intensywności obliczeniowej algorytmu.

3.4. Aproksymacja

3.4.1. Dyskretyzacja obszaru i typy elementów skończonych

Jak wspomniano wcześniej metoda elementów skończonych bazuje na dyskretyzacji rozważanego ciągłego obszaru na określona liczbę elementów będących odwzorowaniami elementów referencyjnych. Elementami tymi zazwyczaj są elementy o prostej geometrii takie jak czworościany, sześciany lub elementy pryzmatyczne. W przypadkach rozważanych w tej pracy korzystano z elementów pryzmatycznych oraz czworościennych (Rys. 3.2).



Rysunek 3.2: Elementy referencyjne użyte w pracy

Dzięki kombinacji tego typu elementów możliwym jest odwzorowanie nawet najbardziej skomplikowanej geometrii obszaru obliczeniowego Ω. W celu określenia poszukiwanych wielkości dla całego obszaru obliczeniowego, a nie tylko w wierzchołkach

elementów, należy określić właściwe funkcje kształtu dla każdego typu elementu zależne od typu oraz stopnia wybranej aproksymacji.

3.4.2. Aproksymacja liniowa

Dla liniowej aproksymacji standardowej (pierwszego rzędu), stopnie swobody elementu związane są z jego wierzchołkami. Bazowe funkcje kształtu dla referencyjnego elementu pryzmatycznego mają postać (3.14):

$$\begin{aligned}
\Phi_0 &= \frac{-z+1}{2}(1-x-y),\\
\Phi_1 &= \frac{-z+1}{2}x,\\
\Phi_2 &= \frac{-z+1}{2}y,\\
\Phi_3 &= \frac{z+1}{2}(1-x-y),\\
\Phi_4 &= \frac{z+1}{2}x,\\
\Phi_5 &= \frac{z+1}{2}y,
\end{aligned} \tag{3.14}$$

gdzie x, y oraz z są współrzędnymi danego punktu. Jak widać są one złożeniem funkcji 2D zależnej od x i y, z funkcją zależną od z. Analogicznie dla referencyjnego elementu czworościennego postać bazowych funkcji kształtu można przedstawić jako (3.15):

$$\begin{aligned}
\Phi_0 &= 1-x-y-z,\\
\Phi_1 &= x,\\
\Phi_2 &= y,\\
\Phi_3 &= z.
\end{aligned} \tag{3.15}$$

W podobny sposób jak rozwiązanie (za pomocą funkcji kształtu) można charakteryzować geometrię elementów. Jak widać z powyższych równań elementy czworościenne charakteryzuje liniowa geometria, pozwalająca na unifikację transformacji jakobianowych dla każdego elementu. W przypadku elementów pryzmatycznych widzimy występowanie wielomianów, które komplikują przekształcenia geometryczne.

3.4.3. Nieciągła dyskretyzacja Galerkina

Dla przykładu zastosowania aproksymacji wyższych rzędów, a co za tym idzie zwiększonego zapotrzebowania na zasoby, wybrano nieciągłą aproksymację Galerkina. W metodzie tej, stopnie swobody elementu związane są z jego wnętrzem, a ich liczba zależy od wybranego stopnia aproksymacji. Zbieżność nieciągłego roz-

Tabela 3.1: Liczba punktów Gaussa oraz funkcji kształtu w zależności od stopnia aproksymacji dla elementu pryzmatycznego

Parametr	**Stopień aproksymacji p**						
	1	*2*	*3*	*4*	*5*	*6*	*7*
N_Q	6	18	48	80	150	231	336
N_S	6	18	40	75	126	196	288

wiązania przybliżonego do ciągłego rozwiązania dokładnego uzyskuje się poprzez wprowadzenie dodatkowych wyrazów w sformułowaniu słabym, zawierających całki po bokach elementów wewnątrz obszaru obliczeniowego, gdzie całkowany jest skok rozwiązania pomiędzy dwoma elementami [15, 16, 17]. W ujęciu przedstawionym w niniejszej pracy całki te traktowane są podobnie jak całki związane z warunkami brzegowymi w sformułowaniach standardowych. Omawiane w pracy całki po wnętrzach elementów są identyczne dla przypadku ciągłej i nieciągłej dyskretyzacji. Dla celów badań wybrano dwa zadania – równanie Poissona oraz uogólnione zadanie konwekcji-dyfuzji.

Dla badanego elementu referencyjnego typu pryzmatycznego, liczba funkcji kształtu przedstawia się wzorem $\frac{1}{2}(p+1)^2(p+2)$. Liczba potrzebnych punktów Gaussa rośnie wraz ze wzrostem stopnia aproksymacji w celu utrzymania zbieżności rozwiązania [14] i dla trójwymiarowej pryzmy jest rzędu $O(p^3)$, tak jak w przypadku liczby funkcji kształtu. Podstawowe paramenty Algorytmu 1 w zależności od stopnia aproksymacji przedstawia tabela 3.1.

3.5. Algorytmy całkowania numerycznego dla zadań testowych

W przypadku obu badanych aproksymacji rozważano zadanie Poissona oraz uogólnione zadanie konwekcji-dyfuzji. W obu przypadkach składowe macierzy sztywności są liczbami skalarnymi więc pętla po N_E obecna w Algorytmie 1 jest pomijana. W przypadku aproksymacji liniowej uwzględniono dwa typy elementów referencyjnych – pryzmatyczne oraz czworościenne. Zadania z nieciągłą aproksymacją Galerkina, jako przykład bardziej wymagających obliczeniowo, przetestowano dla bardziej skomplikowanych elementów typu pryzmatycznego. W zadaniu tym analizowane jest tworzenie macierzy sztywności z pominięciem wektora prawej strony, którego postać zależy od konkretnego zadania (linijki 22-26 Algorytmu 1).

Jak zauważono wcześniej, Algorytm 1 składa się z kilku zagnieżdżonych pętli. W przypadku aproksymacji wyższego rzędu, kwestia którą pętlę zrównoleglić zależy

głównie od dostępnych zasobów sprzętowych. Przy aproksymacji niższych rzędów, najbardziej optymalnym sposobem zrównoleglenia obliczeń, wydaje się podział najbardziej zewnętrznej pętli po elementach. Z tego względu, wewnętrzna organizacja algorytmu może być modyfikowana w zależności od potrzeb oraz użytego typu elementu.

Pierwsza możliwość modyfikacji Algorytmu 1 jest związana z typem używanych elementów. W elementach geometrycznie liniowych (takich jak czworościany), wyrazy jakobianowe są stałe względem całego elementu. W innego typu elementach (np. pryzmatycznych), wszystkie obliczenia jakobianowe są różne dla każdego punktu całkowania. Z tego względu można wyodrębnić dwa warianty algorytmu, osobny dla elementów geometrycznie liniowych, oraz inny dla elementów geometrycznie wieloliniowych. W pierwszym z nich, wszystkie obliczenia dotyczące pochodnych geometrycznych funkcji kształtu oraz ich transformacji, mogą być przeniesione poza pętle po punktach całkowania.

Kolejną optymalizacją algorytmu zastosowaną na etapie projektowania, może być zamiana kolejności pętli po punktach całkowania, z pętlami po funkcjach kształtu. Pozwoli to na bezpośrednią analizę przydatności tego typu rozwiązań, bez konieczności korzystania z opcji kompilatorów dotyczących tego typu optymalizacji.

Ze względu na wymienione wyżej modyfikacje możemy wyróżnić aż 6 wariantów Algorytmu 1. Dodatkowo każdy z nich jest uruchamiany dla dwóch omówionych wcześniej problemów, a co za tym idzie z różną postacią pętli po indeksach i_D, j_D. Ze względu na organizację pętli poszczególne algorytmy (2, 3, 4, 5, 6, 7) zostały nazwane QSS, SQS oraz SSQ wraz z przedrostkami *ogólny* oraz *liniowy* w związku z typem elementu.

Algorytm 2: Wersja algorytmu całkowania numerycznego – *QSS-ogólny*

```
1  for i_Q = 1 to N_Q do
2  |   oblicz ∂ξ/∂x oraz vol (oraz jeżeli to konieczne C);
3  |   for i_S = 1 to N_S do
4  |   |   oblicz globalne pochodne funkcji kształtu w punkcie całkowania;
5  |   end
6  |   for i_S = 1 to N_S do
7  |   |   for j_S = 1 to N_S do
8  |   |   |   A^e[i_S][j_S] += vol ×C[i_Q[[i_D][j_D] × φ[i_Q][i_S][i_D] × φ[i_Q][j_S][j_D];
9  |   |   end
10 |   |   b^e[i_S] += vol ×D[i_Q][i_D] × φ[i_Q][i_S][i_D];
11 |   end
12 end
13 zapisz A^e oraz b^e jako wynik procedury;
```

W celu dokładnego przeanalizowania wszystkich wersji badanego algorytmu dla obu typów aproksymacji oraz wszystkich badanych zadań, należy oszacować ilość

Algorytm 3: Wersja algorytmu całkowania numerycznego – *QSS-liniowy*

oblicz $\frac{\partial \boldsymbol{\xi}}{\partial \boldsymbol{x}}$ oraz **vol**;
for $i_S = 1$ *to* N_S **do**
  oblicz globalne pochodne funkcji kształtu;
end
for $i_Q = 1$ *to* N_Q **do**
  (oblicz jeżeli to konieczne $\boldsymbol{C}$);
  for $i_S = 1$ *to* N_S **do**
    for $j_S = 1$ *to* N_S **do**
      $\boldsymbol{A}^e[i_S][j_S] += \mathbf{vol} \times \boldsymbol{C}[i_Q[[i_D][j_D] \times \boldsymbol{\phi}[i_Q][i_S][i_D] \times \boldsymbol{\phi}[i_Q][j_S][j_D]$;
    end
    $\boldsymbol{b}^e[i_S] += \mathbf{vol} \times \boldsymbol{D}[i_Q][i_D] \times \boldsymbol{\phi}[i_Q][i_S][i_D]$;
  end
end
zapisz $\boldsymbol{A}^e$ oraz $\boldsymbol{b}^e$ jako wynik procedury;

Algorytm 4: Wersja algorytmu całkowania numerycznego – *SQS-ogólny*

for $i_S = 1$ *to* N_S **do**
  zainicjalizuj wiersz macierzy $\boldsymbol{A}^e$;
  for $i_Q = 1$ *to* N_Q **do**
    oblicz $\frac{\partial \boldsymbol{\xi}}{\partial \boldsymbol{x}}$ oraz **vol** (oraz jeżeli to konieczne $\boldsymbol{C}$);
    oblicz globalne pochodne funkcji kształtu w punkcie całkowania;
    for $j_S = 1$ *to* N_S **do**
      $\boldsymbol{A}^e[i_S][j_S] += \mathbf{vol} \times \boldsymbol{C}[i_Q[[i_D][j_D] \times \boldsymbol{\phi}[i_Q][i_S][i_D] \times \boldsymbol{\phi}[i_Q][j_S][j_D]$;
    end
    $\boldsymbol{b}^e[i_S] += \mathbf{vol} \times \boldsymbol{D}[i_Q][i_D] \times \boldsymbol{\phi}[i_Q][i_S][i_D]$;
  end
  zapisz wiersz $\boldsymbol{A}^e$ oraz $\boldsymbol{b}^e[i_S]$ w pamięci;
end

Algorytm 5: Wersja algorytmu całkowania numerycznego – *SQS-liniowy*

oblicz $\frac{\partial \boldsymbol{\xi}}{\partial \boldsymbol{x}}$ oraz **vol**;
for $i_Q = 1$ *to* N_Q **do**
  oblicz globalne pochodne funkcji kształtu;
end
for $i_S = 1$ *to* N_S **do**
  zainicjalizuj wiersz macierzy $\boldsymbol{A}^e$;
  for $i_Q = 1$ *to* N_Q **do**
    (oblicz jeżeli to konieczne $\boldsymbol{C}$);
    for $j_S = 1$ *to* N_S **do**
      $\boldsymbol{A}^e[i_S][j_S] += \mathbf{vol} \times \boldsymbol{C}[i_Q[[i_D][j_D] \times \boldsymbol{\phi}[i_Q][i_S][i_D] \times \boldsymbol{\phi}[i_Q][j_S][j_D]$;
    end
    $\boldsymbol{b}^e[i_S] += \mathbf{vol} \times \boldsymbol{D}[i_Q][i_D] \times \boldsymbol{\phi}[i_Q][i_S][i_D]$;
  end
  zapisz wiersz $\boldsymbol{A}^e$ oraz $\boldsymbol{b}^e[i_S]$ w pamięci;
end

Algorytm 6: Wersja algorytmu całkowania numerycznego – *SSQ-ogólny*

for $i_S = 1$ *to* N_S **do**
  for $j_S = 1$ *to* N_S **do**
    $\boldsymbol{A}^e[i_S][j_S] = 0$; $\boldsymbol{b}^e[i_S] = 0$;
    for $i_Q = 1$ *to* N_Q **do**
      oblicz $\frac{\partial \boldsymbol{\xi}}{\partial \boldsymbol{x}}$ oraz **vol** (oraz jeżeli to konieczne $\boldsymbol{C}$);
      oblicz globalne pochodne funkcji kształtu w punkcie całkowania;
      $\boldsymbol{A}^e[i_S][j_S] +=$ **vol** $\times \boldsymbol{C}[i_Q[[i_D][j_D] \times \boldsymbol{\phi}[i_Q][i_S][i_D] \times \boldsymbol{\phi}[i_Q][j_S][j_D]$;
      if $i_S = j_S$ **then**
        $\boldsymbol{b}^e[i_S] +=$ **vol** $\times \boldsymbol{D}[i_Q][i_D] \times \boldsymbol{\phi}[i_Q][i_S][i_D]$;
      end
    end
    zapisz $\boldsymbol{A}^e$ oraz $\boldsymbol{b}^e[i_S]$ (jeżeli $i_S = j_S$) w pamięci;
  end
end

Algorytm 7: Wersja algorytmu całkowania numerycznego – *SSQ-liniowy*

oblicz $\frac{\partial \boldsymbol{\xi}}{\partial \boldsymbol{x}}$ oraz **vol**;
for $i_S = 1$ *to* N_S **do**
  oblicz globalne pochodne funkcji kształtu;
end
for $i_S = 1$ *to* N_S **do**
  for $j_S = 1$ *to* N_S **do**
    $\boldsymbol{A}^e[i_S][j_S] = 0$; $\boldsymbol{b}^e[i_S] = 0$;
    for $i_Q = 1$ *to* N_Q **do**
      (oblicz jeżeli to konieczne $\boldsymbol{C}$);
      $\boldsymbol{A}^e[i_S][j_S] +=$ **vol** $\times \boldsymbol{C}[i_Q[[i_D][j_D] \times \boldsymbol{\phi}[i_Q][i_S][i_D] \times \boldsymbol{\phi}[i_Q][j_S][j_D]$;
      if $i_S = j_S$ **then**
        $\boldsymbol{b}^e[i_S] +=$ **vol** $\times \boldsymbol{D}[i_Q][i_D] \times \boldsymbol{\phi}[i_Q][i_S][i_D]$;
      end
    end
    zapisz $\boldsymbol{A}^e$ oraz $\boldsymbol{b}^e[i_S]$ (jeżeli $i_S = j_S$) w pamięci;
  end
end

operacji oraz dostępów do pamięci. Pozwoli to na wstępna analizę w jaki sposób należy algorytm zoptymalizować dla danego sprzętu i jego (ograniczonych) zasobów obliczeniowych i pamięciowych.

3.6. Złożoność obliczeniowa algorytmu całkowania numerycznego

W przypadku szacowania liczby operacji w poszczególnych wersjach algorytmu całkowania numerycznego możemy wyszczególnić następujące etapy obliczeń:

1. Obliczenia jakobianowe (linijka 9 Algorytmu 1)
   — Obliczenie pochodnych bazowych funkcji kształtu dla elementów pryzmatycznych (wzór (3.14)) - ze względu na powtarzające się obliczenia możemy założyć, iż każdy kompilator dokona optymalizacji redukując liczbę operacji do 14.
   — Macierz J i jej odwrotność, wyznacznik i element objętościowy (**vol**) - 18 operacji dla każdego z geometrycznych stopni swobody dla pryzm oraz 9 operacji dla czworościanów przy tworzeniu macierzy jakobianowej + 43 operacje związane z jej odwrotnością i wyznacznikiem dla obu typów elementów.
2. Obliczenie pochodnych funkcji kształtu względem współrzędnych globalnych (linijka 10 Algorytmu 1) - zazwyczaj wykonywane przed podwójną pętlą po funkcjach kształtu w celu uniknięcia redundantności obliczeń (aczkolwiek można rozważyć wersję algorytmu z pochodnymi obliczanymi wewnątrz pętli po indeksie i_S) – 15 operacji dla każdej funkcji kształtu.
3. Dla zadania konwekcji-dyfuzji – obliczanie iloczynu współczynników macierzy $C[i_Q]$ z funkcjami kształtu zależnymi od pierwszej pętli po funkcjach kształtu (i_S) - optymalizacja redukująca powtarzanie obliczeń poprzez wyjęcie przed pętle j_S wyrazów niezależnych od jej indeksu – 28 operacji dla aproksymacji liniowej oraz 22 operacje dla nieciągłej aproksymacji Galerkina.
4. Obliczenie wektora prawej strony (linijka 24 Algorytmu 1) – 3 operacje dla zadania Poissona oraz 9 operacji dla konwekcji-dyfuzji.
5. Końcowe obliczanie wyrazów macierzy sztywności (linijka 18 Algorytmu 1) – 9 operacji dla konwekcji-dyfuzji oraz 7 dla zadania Poissona.

W przypadku aproksymacji liniowej ostateczne liczby operacji dla poszczególnych etapów całkowania numerycznego należy pomnożyć przez współczynniki zaprezentowane w tabeli 3.2.

Tabela 3.2: Współczynniki mnożenia dla każdego z etapów i poszczególnych wariantów algorytmu całkowania numerycznego z liniową aproksymacją standardową

		Poisson		**konwekcja-dyfuzja**	
		czworościan	*pryzma*	*czworościan*	*pryzma*
QSS	*1*	1	N_Q	1	N_Q
	2	N_S	N_S	N_S	N_S
	3	0	0	$N_Q * N_S$	$N_Q * N_S$
	4	$N_Q * N_S$	$N_Q * N_S$	$N_Q * N_S$	$N_Q * N_S$
	5	$N_Q * N_S * N_S$	$N_Q * N_S * N_S$	$N_Q * N_S * N_S$	$N_Q * N_S * N_S$
SQS	*1*	1	$N_Q * N_S$	1	$N_Q * N_S$
	2	N_S	$N_Q * N_S$	N_S	$N_Q * N_S$
	3	0	0	$N_Q * N_S$	$N_Q * N_S$
	4	$N_Q * N_S$	$N_Q * N_S$	$N_Q * N_S$	$N_Q * N_S$
	5	$N_Q * N_S * N_S$	$N_Q * N_S * N_S$	$N_Q * N_S * N_S$	$N_Q * N_S * N_S$
SSQ	*1*	1	$N_Q * N_S * N_S$	1	$N_Q * N_S * N_S$
	2	N_S	$N_Q * N_S * N_S$	N_S	$N_Q * N_S * N_S$
	3	0	0	$N_Q * N_S * N_S$	$N_Q * N_S * N_S$
	4	$N_Q * N_S * N_S$	$N_Q * N_S * N_S$	$N_Q * N_S * N_S$	$N_Q * N_S * N_S$
	5	$N_Q * N_S * N_S$	$N_Q * N_S * N_S$	$N_Q * N_S * N_S$	$N_Q * N_S * N_S$

W przypadku liniowej aproksymacji standardowej liczby N_Q i N_S są sobie równe i wynoszą odpowiednio 6 dla pryzm oraz 4 dla czworościanów. Podsumowanie wyżej wymienionych szacowań dla liniowej aproksymacji standardowej prezentuje tabela 3.3.

Jak można zauważyć z tabeli algorytmy SQS oraz SSQ dla pryzm powodują dużą ilość redundantnych obliczeń, co może mieć wpływ na wydajność. Należy jednakże zauważyć, że w przypadku programowania akceleratorów, duża liczba wykonywanych operacji może stanowić istotną zaletę wpływającą na nasycenie maszyny odpowiednią ilością obliczeń i korzystnym stosunkiem liczby obliczeń do liczby dostępów w pamięci.

Finalna liczba operacji wykonywanych przez procesory dla każdej wersji algorytmu będzie zależała od zastosowanych optymalizacji, zarówno ręcznych jak i automatycznych, dokonywanych przez kompilator. Ze względu na fakt stałości niektórych współczynników (np. pochodnych funkcji kształtu dla wszystkich punktów całkowania dla elementów czworościennych, stałych współczynników problemowych dla zadania konwekcji-dyfuzji-reakcji) oraz fakt symetryczności otrzymanych macierzy, niektóre kompilatory są w stanie znacznie zredukować szacowaną liczbę operacji. Tak samo, w związku z tym, iż ostateczne obliczenia wykonywane są w potrójnej pętli po punktach całkowania oraz funkcjach kształtu, kompilatory są w stanie dokonać rozwinięcia tychże pętli, co może spowodować że liczba operacji w nich wykonana

będzie taka sama niezależnie od wariantu algorytmu. Szczególnie prawdopodobne może to być dla elementów czworościennych, gdzie duża liczba niezależnych obliczeń może zostać przeniesiona poza wyżej wymienione pętle. Dla elementów pryzmatycznych oraz zadania konwekcji-dyfuzji oszacowania z tabeli 3.3 powinny odpowiadać rzeczywistej liczbie operacji wykonywanych przez sprzęt.

Tabela 3.3: Szacowana na podstawie wzorów ogólnych liczba operacji dla poszczególnych wariantów całkowania numerycznego dla liniowej aproksymacji standardowej

		Poisson		**konwekcja-dyfuzja**	
		czworościan	*pryzma*	*czworościan*	*pryzma*
QSS	*1*	52	990	52	990
	2	60	540	60	540
	3	0	0	448	1008
	4	48	108	144	324
	5	448	1512	576	1944
	$\sum$	***608***	***3150***	***1280***	***4806***
SQS	*1*	52	5940	52	5940
	2	60	3240	60	3240
	3	0	0	448	1008
	4	48	108	144	324
	5	448	1512	576	1944
	$\sum$	***608***	***10800***	***1280***	***12456***
SSQ	*1*	52	35640	52	35640
	2	60	19440	60	19440
	3	0	0	1792	6048
	4	192	648	576	864
	5	448	1512	576	1944
	$\sum$	***752***	***57240***	***3056***	***63936***

Tabela 3.4: Szacowana liczba operacji dla zadania konwekcji-dyfuzji z nieciągłą aproksymacją Galerkina

	Stopień aproksymacji p						
	1	*2*	*3*	*4*	*5*	*6*	*7*
1	990	2970	7920	13200	24750	38115	55440
2	540	4860	28800	90000	283500	679140	1451520
3	792	7128	42240	132000	415800	996072	2128896
5	1944	52488	691200	4050000	21432600	79866864	250822656
$\sum$	***4266***	***67446***	***770160***	***4285200***	***22156650***	***81580191***	***254458512***

W zadaniach z nieciągłą aproksymacją Galerkina korzystano z algorytmu z organizacją pętli QSS oraz z elementów pryzmatycznych. Jak również wspomniano wcześniej, w tym rodzaju aproksymacji pominięto tworzenie wektora obciążeń. Korzystając z parametrów N_Q i N_S zaprezentowanych w tabeli 3.1 oraz współczynników

z odpowiednich kolumn tabeli 3.2 dokonano obliczeń liczby operacji w zależności od stopnia aproksymacji. Wyniki zaprezentowano w tabeli 3.4.

Kolejnym aspektem wymagającym analizy jest rozmiar danych potrzebnych do przechowywania bieżących obliczeń. Dla aproksymacji liniowej rozmiar potrzebnych danych przedstawia tabela 3.6. Jak można zauważyć, liczba używanych wartości zmienia się w zakresach od kilku liczb do kilkuset. Może to być kluczowym kryterium dotyczącym wyboru odpowiedniego algorytmu na daną architekturę. Najmniejsze wymagania dotyczące zasobów ma wersja SSQ. Jednakże należy wziąć pod uwagę fakt, że np. dla elementów pryzmatycznych liczba potrzebnych operacji gwałtownie rośnie ze względu na powtarzające się obliczenia jakobianowe. Z drugiej strony wersja QSS, w której unikamy powtarzania obliczeń, ma bardzo duże wymagania odnośnie zasobów. Wersja SQS rozważana niejako jako wersja pomiędzy poprzednimi, charakteryzuje się tym że wymaga przechowywania tylko jednego wiersza macierzy oraz powtarzania obliczeń jakobianowych tylko dla niego. Dla algorytmu całkowania numerycznego z nieciągłą aproksymacją Galerkina rozmiar potrzebnych danych zwiększa się wraz ze wzrostem stopnia aproksymacji (Tab. 3.5). W tabeli przedstawiono dane dla zadania konwekcji-dyfuzji, gdyż w przypadku zadania Poissona wystarczy odjąć rozmiar współczynników problemowych.

W praktyce, ze względu na różnice w ostatecznej optymalizacji kodu, jako jedyne do badań nad modelowym czasem wykonania kodu stosowane były liczby operacji dla elementów pryzmatycznych i zadania konwekcji-dyfuzji jako najmniej podlegające optymalizacjom redukującym liczbę operacji wykonywanych przez procesory. Dodatkowym uzasadnieniem tego ograniczenia jest fakt, że ze względu na intensywność arytmetyczną różnych wariantów kodu, tylko w tym przypadku wydajność obliczeń może mieć, dla większości architektur, istotniejsze znaczenie od wydajności pamięci RAM.

Tabela 3.5: Wymagania pamięciowe dla całkowania numerycznego w zadaniu konwekcji-dyfuzji oraz nieciągłej aproksymacji Galerkina

	Stopień aproksymacji p						
	1	*2*	*3*	*4*	*5*	*6*	*7*
Dane całkowania	24	72	192	320	600	924	1344
Dane we/wy dla każdego elementu skończonego							
Współczynniki $\boldsymbol{C}$	16	16	16	16	16	16	16
Dane geometryczne	18	18	18	18	18	18	18
Macierz $\boldsymbol{A}^e$	36	324	1600	5625	15876	38416	82944
$\sum$ *(minimalna liczba dostępów do RAM)*	**70**	**358**	**1634**	**5659**	**15910**	**38450**	**82978**
ϕ	144	1296	7680	24000	75600	181104	387072
$\sum$	***238***	***1726***	***9506***	***29979***	***92110***	***220478***	***471394***

Tabela 3.6: Podstawowe parametry algorytmu całkowania numerycznego, wraz z wymaganiami pamięciowymi dla różnych wersji procedury

	Poisson		**konwekcja-dyfuzja**	
	czworościan	*pryzma*	*czworościan*	*pryzma*
Dane całkowania (element referencyjny)	$N_Q + 3 = 7$	$4 * N_Q = 24$	$N_Q + 3 = 7$	$4 * N_Q = 24$
Dane we/wy dla każdego elementu skończonego				
Dane geometryczne (we)	12	18	12	18
Współczynniki problemowe (we)	$1 * N_Q = 4$	$1 * N_Q = 6$	20	20
Macierz sztywności $\boldsymbol{A}^e$ *(wy)*	$N_S * N_S = 16$	$N_S * N_S = 36$	$N_S * N_S = 16$	$N_S * N_S = 36$
Wektor obciążeń $\boldsymbol{b}^e$ *(wy)*	$N_S = 4$	$N_S = 6$	$N_S = 4$	$N_S = 6$
Sumarycznie (minimalna liczba dostępów do pamięci RAM)	**36**	**66**	**52**	**80**
Dla pojedynczego punktu całkowania – wersja QSS				
Współczynniki problemowe $\boldsymbol{C}$ *i* $\boldsymbol{D}$	1	1	20	20
Funkcje kształtu i ich pochodne $\boldsymbol{\phi}$	$4 * N_S = 16$	$4 * N_S = 24$	$4 * N_S = 16$	$4 * N_S = 24$
Sumarycznie (razem z $\boldsymbol{A}^e$ *oraz* $\boldsymbol{b}^e$*)*	**37**	**67**	**56**	**86**
Dla wszystkich punktów całkowania – wersja SQS				
Współczynniki problemowe $\boldsymbol{C}$ *i* $\boldsymbol{D}$	$1 * N_Q = 4$	$1 * N_Q = 6$	20	20
Funkcje kształtu i ich pochodne $\boldsymbol{\phi}$	$3 * N_S + 1 * N_Q * N_S = 28$	$4 * N_S * N_Q = 144$	$3 * N_S + 1 * N_Q * N_S = 28$	$4 * N_S * N_Q = 144$
Sumarycznie (razem z $\boldsymbol{A}^e$ *oraz* $\boldsymbol{b}^e$*)*	**52**	**192**	**68**	**206**
Dla wszystkich punktów całkowania – wersja SSQ				
Współczynniki problemowe $\boldsymbol{C}$ *i* $\boldsymbol{D}$	$1 * N_Q = 4$	$1 * N_Q = 6$	20	20
Funkcje kształtu i ich pochodne $\boldsymbol{\phi}$	4	4	8	8
Sumarycznie (razem z jednym elementem z $\boldsymbol{A}^e$ *oraz* $\boldsymbol{b}^e$*)*	**10**	**12**	**30**	**30**

3.7. Intensywność arytmetyczna

Istotnym elementem analizy wydajności jest badanie liczby dostępów do różnych poziomów hierarchii pamięci w trakcie wykonywania algorytmu. Badanie takie jest możliwe tylko przy uwzględnieniu szczegółów realizacji obliczeń na konkretnych architekturach. W specyficznych przypadkach możliwa jest ograniczona analiza, pozwalająca na scharakteryzowanie relacji między wymaganiami algorytmu odnośnie liczby operacji oraz liczby dostępów do pamięci.

Przypadkiem takim jest np. liniowa aproksymacja standardowa, gdzie rozmiary tablic wykorzystywanych w obliczeniach są relatywnie małe. Dla procesorów z odpowiednio dużą pamięcią podręczną, można wtedy założyć, że dla każdego elementu procesor pobiera z pamięci DRAM potrzebne dane wejściowe, wszystkie obliczenia wykonuje na tablicach i zmiennych przechowywanych w rejestrach i pamięci podręcznej (dla procesorów GPU także w pamięci wspólnej), a na zakończenie pracy zapisuje wynik, w postaci elementowej macierzy sztywności i wektora prawej strony, ponownie w pamięci DRAM.

Tak określona minimalna liczba dostępów do pamięci DRAM może służyć do obliczenia intensywności arytmetycznej algorytmu, wyrażanej w liczbie operacji na pojedynczy dostęp do pamięci. Intensywność arytmetyczna jest istotnym parametrem wydajnościowym - od jej wartości zależy, czy procesor w swojej pracy będzie miał zawsze dostępne potrzebne dane (dla wysokich wartości intensywności), czy też jego potoki przetwarzania będą oczekiwały na dostarczenie danych z pamięci (dla niskich wartości intensywności).

To, która z powyższych możliwości wystąpi w konkretnych obliczeniach zależy od stosunku intensywności arytmetycznej algorytmu do charakterystyk procesora. Dla każdego z procesorów można obliczyć graniczną wartość intensywności arytmetycznej, oddzielającą przypadki ograniczenia wydajności przez możliwości przetwarzania potoków konkretnych operacji np. zmiennoprzecinkowych (w przypadku kiedy intensywność jest wysoka i potoki nie oczekują na dane) od przypadków, gdy wydajność jest ograniczana przez układ pamięci (dane z pamięci nie są dostarczane wystarczająco szybko, żeby zapewnić pracę potoków przetwarzania bez przestojów – dla zbyt niskiej intensywności). Tak określona wartość graniczna intensywności nazywana bywa wartością równowagi procesora ("processor balance"). Dla każdego algorytmu, dla którego da się oszacować liczbę wykonywanych operacji i liczbę dostępów do pamięci, można określić czy wydajność konkretnego procesora (przynajmniej teoretycznie) jest ograniczana przez wydajność potoków przetwarzania, czy przez układ pamięci.

Tabela 3.7: Graniczna intensywność arytmetyczna badanych wersji algorytmu całkowania numerycznego dla aproksymacji liniowej

	Poisson		**konwekcja-dyfuzja**	
	czworościan	*pryzma*	*czworościan*	*pryzma*
QSS	16,89	47,73	24,62	60,08
SQS	16,89	163,64	24,62	155,70
SSQ	20,89	867,27	58,77	799,20

Tabela 3.8: Graniczna intensywność arytmetyczna badanego algorytmu całkowania numerycznego dla nieciągłej aproksymacji Galerkina

Stopień aproksymacji p						
1	*2*	*3*	*4*	*5*	*6*	*7*
60,94	188,40	471,33	757,24	1392,62	2121,72	3066,58

Dla szacowanych liczb operacji zmiennoprzecinkowych i dostępów do pamięci, w przypadku całkowania dla aproksymacji liniowej wartości intensywności arytmetycznej zostały zaprezentowane w tabeli 3.7.

W taki sam sposób obliczono wartość intensywności arytmetycznej dla poszczególnych stopni aproksymacji w przypadku całkowania numerycznego w zadaniu konwekcji-dyfuzji z nieciągłą aproksymacją Galerkina 3.8.

W oszacowaniach przyjęto liczbę operacji, która może być uznana za maksymalną (tzn. bez uwzględnienia szeregu możliwych optymalizacji) oraz minimalną liczbę dostępów do pamięci RAM (tylko dane wejściowe i wyjściowe algorytmu). Tak więc w rzeczywistych obliczeniach intensywność może być niższa. Jak widać w przypadku elementów pryzmatycznych i zadania konwekcji dyfuzji intensywność jest najwyższa, a jak było to już wspomniane, jej wartość w praktyce nie powinna być znacząco niższa.

3.8. Podsumowanie

Przedstawione w niniejszym rozdziale modelowe problemy oraz opracowane dla nich warianty algorytmu całkowania numerycznego, pozwalają spojrzeć na badaną procedurę jako bardzo szczegółowy tester (benchmark) możliwości obliczeniowych różnorodnego sprzętu. Dzięki manewrowaniu poszczególnymi parametrami badanych zadań, jesteśmy w stanie sprawdzić czy posiadany przez nas sprzęt pasuje do poszczególnego zadania. W dalszych rozważaniach autor dokonał szczegółowych badań prezentowanych wariantów algorytmu dla różnych architektur sprzętowych, co pozwoliło na porównanie wydajności, rozważanych w tymże rozdziale, z warto-

ściami empirycznymi. Dokonał także analizy określającej odpowiednie parametry testowanego sprzętu, mające wpływ na wydajność badanego algorytmu.

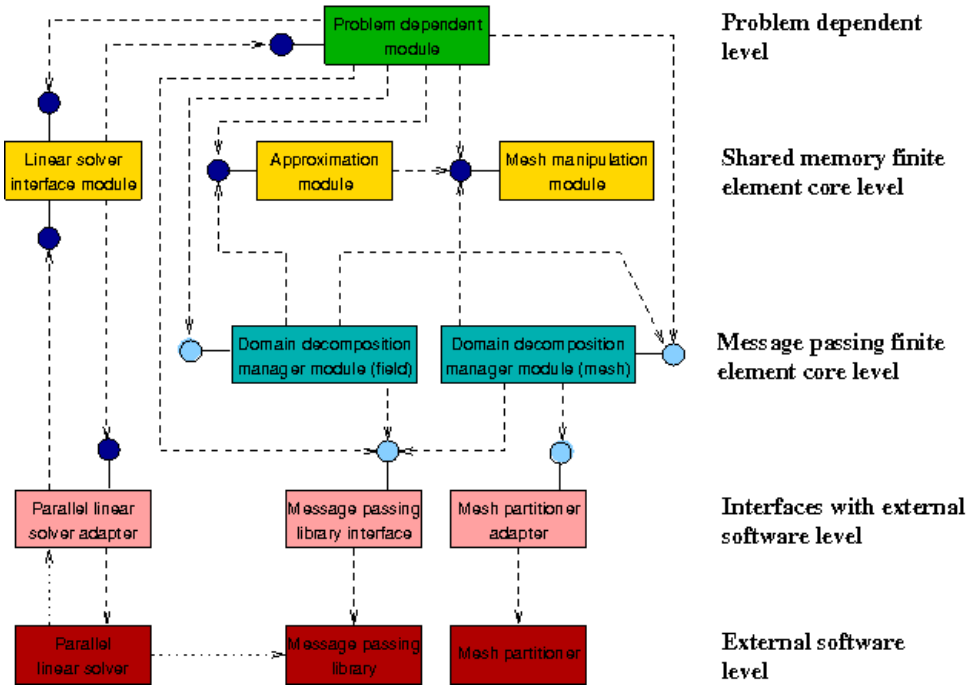
Rozdział 4

Model programowania

4.1. Wykorzystywane narzędzia programistyczne

4.1.1. Program symulacji metodą elementów skończonych

Jako podstawowe narzędzie wykorzystywane w badaniach, użyto modularnego szkieletu programistycznego do obliczeń inżynierskich przy pomocy Metody Elementów Skończonych – ModFEM [87]. Dzięki swojej modularnej budowie, pozwala on na modyfikację poszczególnych fragmentów obliczeń MES takich jak aproksymacja, obsługa siatki czy solwery rozwiązań (Rys. 4.1).



Rysunek 4.1: Schemat szkieletu ModFEM

Jak widać na rysunku 4.1, program składa się z kilku poziomów na których są obecne poszczególne moduły. Głównym modułem zarządzanym przez użytkownika końcowego jest moduł problemowy, który służy do zdefiniowania sformułowania

słabego MES oraz określa z jakich innych modułów użytkownik korzysta. Najważniejsze moduły zostały zaznaczone kolorem żółtym i są to:

- moduł siatki – moduł zarządzający operacjami na siatce, typem elementów oraz pozwalający na operacje adaptacji i deadaptacji poszczególnych elementów.
- moduł aproksymacji – pozwala na dokonywanie niezbędnych w MES procedur związanych z typem aproksymacji takich jak całkowanie numeryczne czy projekcja rozwiązania.
- moduł solwera – pozwala na wykorzystywanie różnego typu solwerów liniowych do ostatecznego rozwiązania utworzonego układu równań liniowych.

Pozostałe moduły w kodzie ModFEM służą do różnego typu podziału pracy na różnorodnego rodzaju sprzęcie. Jak można zauważyć, framework ten pozwala na pracę w środowisku rozproszonym, co czyni go doskonałym narzędziem zarówno do obliczeń MES na akceleratorach i pojedynczych komputerach jak i w środowisku obliczeń wysokiej wydajności takich jak np. klastry lub specjalistyczne superkomputery.

W ramach badań opracowano rozszerzenie modułu aproksymacji o odpowiednią obsługę akceleratorów. Zmodyfikowano także moduły problemowe badanych zagadnień, w celu odpowiedniego przygotowania struktur danych oraz porównawczego przetestowania algorytmu całkowania numerycznego w środowisku OpenMP z odpowiednimi optymalizacjami. Szkielet ModFEM został również uzupełniony o moduł z funkcjami wspólnymi dla obsługi języków CUDA i OpenCL – inicjalizacją, tworzeniem kontekstu i innymi funkcjonalnościami ułatwiającymi pracę w tych środowiskach.

4.1.2. Języki programowania i rozszerzenia do obliczeń wielowątkowych

Językiem programowania używanym do tworzenia kodu był język C, jako jeden z najbardziej wydajnych języków wysokiego poziomu, pozwalający równocześnie na stosunkowo niskopoziomową kontrolę nad sprzętem i tym, w jaki sposób interpretuje on wydawane rozkazy. Język C jest również wzorem na którym oparto języki programowania akceleratorów CUDA oraz OpenCL.

Podstawowym kompilatorem używanym przez autora był kompilator Intel C Compiler [49] będący częścią pakietu Intel Parallel Studio XE oraz kompilator GNU C Compiler (gcc) wraz z rozszerzeniami wspierającymi obsługę równoległości oraz wektoryzacji. W trakcie programowania procesorów ogólnego przeznaczenia autor wykorzystywał następujące biblioteki oraz rozszerzenia języka:

OpenMP – interfejs programowania aplikacji pozwalający na stosowanie wątków w modelu pamięci wspólnej. Pozwala on na podział pracy pomiędzy poszczególne wątki w zależności od potrzeb. Posiada wbudowane konstrukcje ułatwiające podział pracy, dzięki czemu można równoważyć obciążenie w zależności od dostępnych zasobów [97]

Intel Cilk Plus – rozszerzenia języka C wprowadzające możliwości zrównoleglenia na sposób analogiczny do OpenMP. Dodatkowo Cilk Plus wprowadza notację tablicową (Intel Cilk Plus Array Notation), która ułatwia obsługę tablic dla programisty oraz automatyczną wektoryzację dla kompilatora (Rys. 4.2). Dzięki rozszerzeniom języka, Intel Cilk Plus pozwala na bezpośrednie określenie fragmentów do zwektoryzowania przez kompilator, oraz pozwala na wyrównanie tablic w celu dopasowania do rejestrów wektorowych.

Notacja standardowa:

```
for(i=0; i<STR; i++)
        temp[i] =  shpx[i] * jac_0 + shpy[i];
```

Notacja Intel Cilk Plus:

```
temp[0:STR]  =  shpx[0:STR] * jac_0 + shpy[0:STR];
```

Rysunek 4.2: Notacja tablicowa Cilk Plus (fragment linii 10 w Algorytmie 1)

Intel Intrinsics – zestaw instrukcji mający bezpośrednie przełożenie na język maszynowy (assembler) danego procesora. Pozwalają one na całkowitą kontrolę nad sposobem wykonania, dając równocześnie pełen dostęp do jednostek arytmetyczno-logicznych procesora. Pozwala to na bezpośrednią optymalizację kodu, bez potrzeby korzystania z opcji kompilatora (Rys. 4.3).

Intel Intrinsic:

```
shp1=_mm256_mul_pd(coeff03,shp1);
shp2=_mm_mul_pd(_mm256_castpd256_pd128(coeff03),shp2);
coeff03 = _mm256_set1_pd(vol);
shp1=_mm256_mul_pd(coeff03,shp1);
load_vec1=_mm256_add_pd(load_vec1,shp1);
shp2=_mm_mul_pd(_mm256_castpd256_pd128(coeff03),shp2);
load_vec2=_mm_add_pd(load_vec2,shp2);
```

Intel Cilk Plus:

```
load_vec[0:NSHAP] += (coeff03 * shp[0:NSHAP])*vol;
```

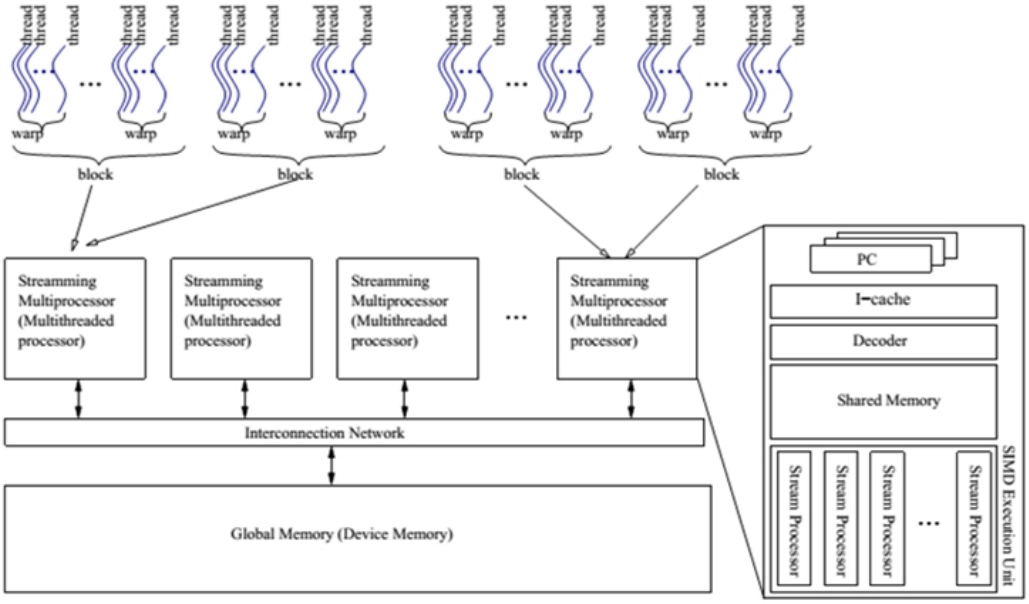
Rysunek 4.3: Instrukcje Intel Intrinsics

Na rysunku widać różnice pomiędzy notacją Cilk (po prawej) oraz instrukcjami Intrinsics (po lewej) podczas obliczania wektora prawej strony (linijki 23-24 Algorytmu 1).

4.1.3. Programowanie akceleratorów

4.1.3.1. CUDA

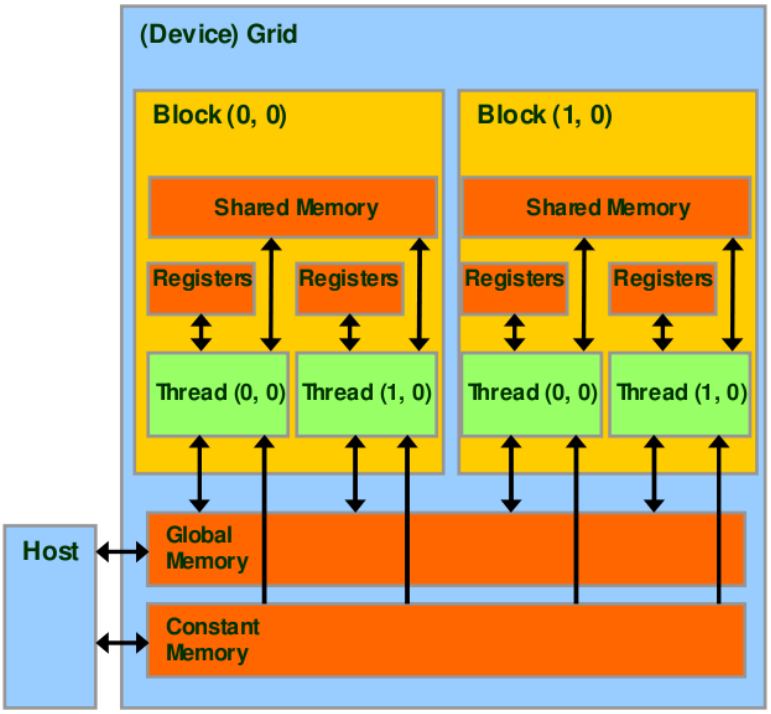
Podstawowymi narzędziami programistycznymi wykorzystanymi do programowania akceleratorów były interfejsy programowania aplikacji CUDA oraz OpenCL. Pierwszy z nich, stworzony został przez firmę Nvidia jako narzędzie wspomagające programowanie kart graficznych i jako taki, wzoruje się bezpośrednio na budowie sprzętu [94]. Model wykonania CUDA bazuje na podziale obliczeń pomiędzy wątki (threads), które grupowane są w, opisywane wcześniej w rozdziale 2.2 warpy, następnie w bloki (blocks), a ostatecznie w siatkę (grid). Odpowiada to bezpośrednio budowie akceleratora graficznego, gdzie GPU może być traktowany jako siatka multiprocesorów podzielonych na mniejsze bloki pojedynczych jednostek przetwarzania (Rys. 4.4).



Rysunek 4.4: Mapowanie CUDA na architekturę [94]

Tak samo jak w przypadku modelu wykonania, schemat pamięci, bazuje na najczęściej spotykanym w kartach graficznych podziale na pamięć wspólną (shared memory) dla wątków pracujących w ramach jednego bloku, pamięć globalną urządzenia (global memory) oraz bufor pamięci stałej (constant memory). Dla każdego wątku dostępna jest odpowiednia ilość rejestrów w zależności od fizycznych zasobów multiprocesora oraz zdefiniowanego podziału pracy między siatki oraz bloki (Rys. 4.5).

CUDA definiuje zbiór bibliotek, dyrektyw i funkcji dla języków C/C++ oraz Fortran, a także dostarcza odpowiednie narzędzia do ich obsługi, takie jak kompila-



Rysunek 4.5: Model pamięci CUDA [21]

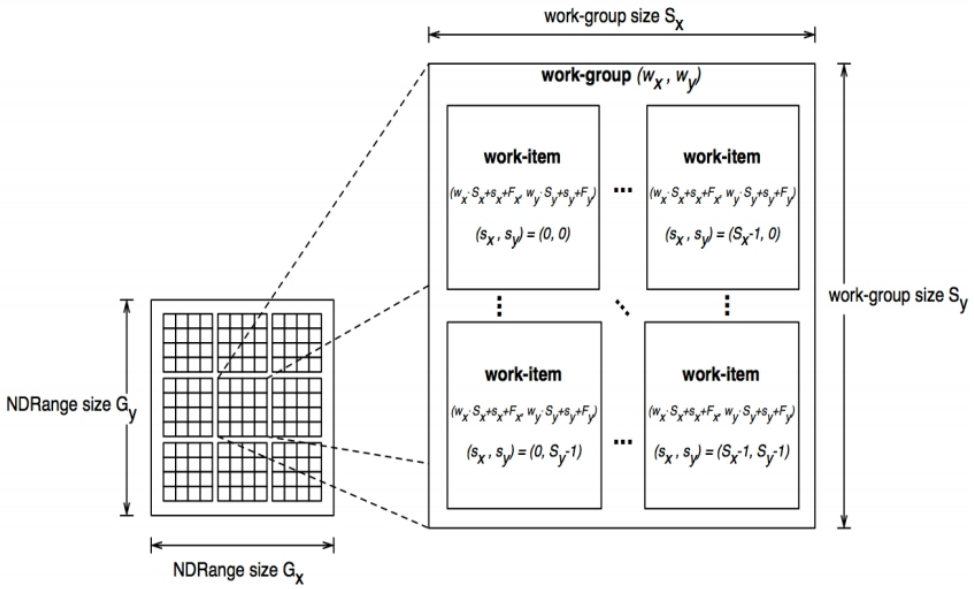
tory, debuggery oraz profilery. Fragmenty kodu wykonywanego na procesorze mają postać funkcji zwanych kernelami. Maszyna na której dostępny jest akcelerator nazywana jest gospodarzem (host) i zazwyczaj jest definiowana jako procesor ogólnego przeznaczenia (CPU) wraz z dostępną mu pamięcią (RAM).

Głównym ograniczeniem platformy programistycznej CUDA jest jej zamknięty standard, pozwalający na użycie jedynie na rozwiązaniach firmy Nvidia. Z tego względu lepszym rozwiązaniem pozwalającym na przenośność kodu pomiędzy architekturami różnych producentów jest standard OpenCL.

4.1.3.2. OpenCL

OpenCL opracowany został w 2008 roku przez korporację Khronos Group, która powstała jako konsorcjum zrzeszające głównych producentów sprzętu i oprogramowania do przetwarzania grafiki, takich jak Apple, Intel, SGI, Nvidia oraz ATI. W swojej budowie oparty jest on na takim samym wzorcu jak CUDA, jednakże jego standard jest otwarty i pozwala twórcom sprzętu na przygotowanie swojej własnej implementacji na daną architekturę. OpenCL pozwala na programowanie praktycznie każdego rodzaju wielordzeniowych i wektorowych maszyn - od nowoczesnych CPU aż do GPU, przez jednostki hybrydowe PowerXCell, APU oraz akceleratory Intel Xeon Phi. Specyfikacja OpenCL zawiera w sobie język programowania, bazują-

cy na standardzie C99, służący do programowania akceleratorów, oraz interfejs programowania aplikacji (API) do obsługi platformy (rozumianej jako dana kombinacja dostępnego sprzętu obliczeniowego i oprogramowania systemowego) i uruchamiania zadań na procesorach [41]. Każde z możliwych do wykorzystania urządzeń, jest określane w modelu OpenCL jako *Urządzenie Obliczeniowe* (*Compute Device*), a każdy z jego rdzeni/multiprocesorów jako *Jednostka Obliczeniowa* (*Compute Unit*). Analogicznie do modelu CUDA, OpenCL definiuje pojedyncze wątki jako elementy robocze (work-item), które są pogrupowane w grupy robocze (work-group) (Rys. 4.6).

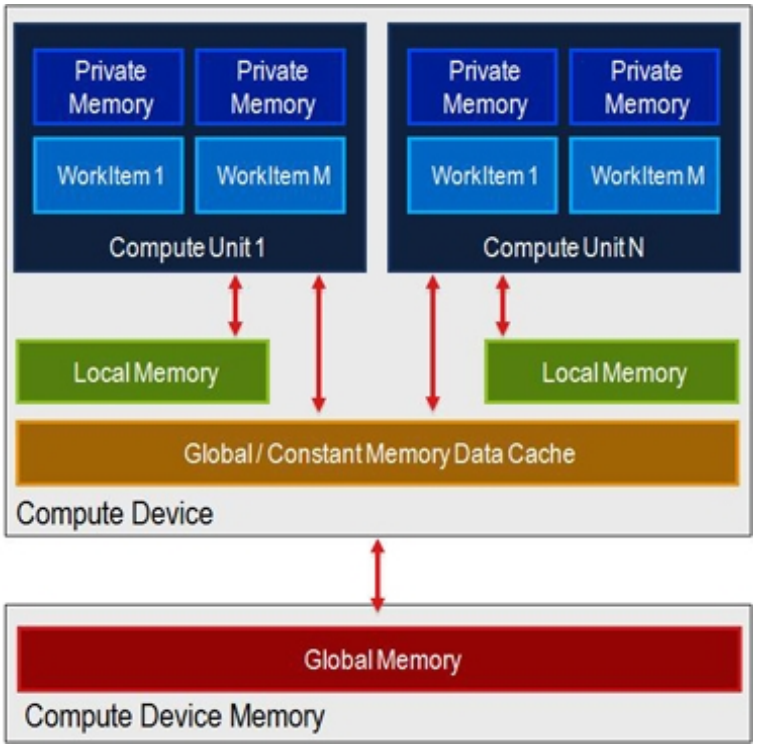


Rysunek 4.6: Model wykonania OpenCL [41]

Wszystkie grupy robocze uruchamiane są na urządzeniu jako n-wymiarowy zakres obliczeń (NDRange). Optymalny rozmiar grupy roboczej jest zależny od architektury i zazwyczaj jest związany z jej wewnętrzną budową, czyli np. ilością procesorów w multiprocesorze lub szerokością rejestrów wektorowych.

Tak samo, model pamięci jest analogiczny do tego spotykanego w CUDA i różni się jedynie nazewnictwem (pamięć wspólna – pamięć lokalna, rejestry – pamięć prywatna) (Rys. 4.7).

Ze względu na przenośność kodu OpenCL pomiędzy urządzeniami różnych typów, każdy z obszarów pamięci może być fizycznie inaczej odwzorowywany w zależności od dostępnych zasobów sprzętowych. Powoduje to trudności z odpowiednim odwzorowywaniem sprzętu podczas programowania architektur o budowie innej niż standardowe GPU. Kolejnym problemem z tym związanym jest to, że prawdziwe odwzorowanie kodu na sprzęt jest ukryte przed programistą, co pozbawia go pełnej



Rysunek 4.7: Model pamięci OpenCL [41]

kontroli nad wykonaniem programu. OpenCL zawiera procedury pozwalające zachować przenośność kodu pomiędzy rożnymi platformami sprzętowymi przez odpowiednie dostosowanie swojego modelu pamięci oraz wykonania [107]. Bezpośrednia implementacja na danej maszynie, zależna jest od dostawcy platformy programistycznej OpenCL oraz odpowiednich sterowników.

4.1.3.3. Komunikacja pamięć hosta – pamięć akceleratora

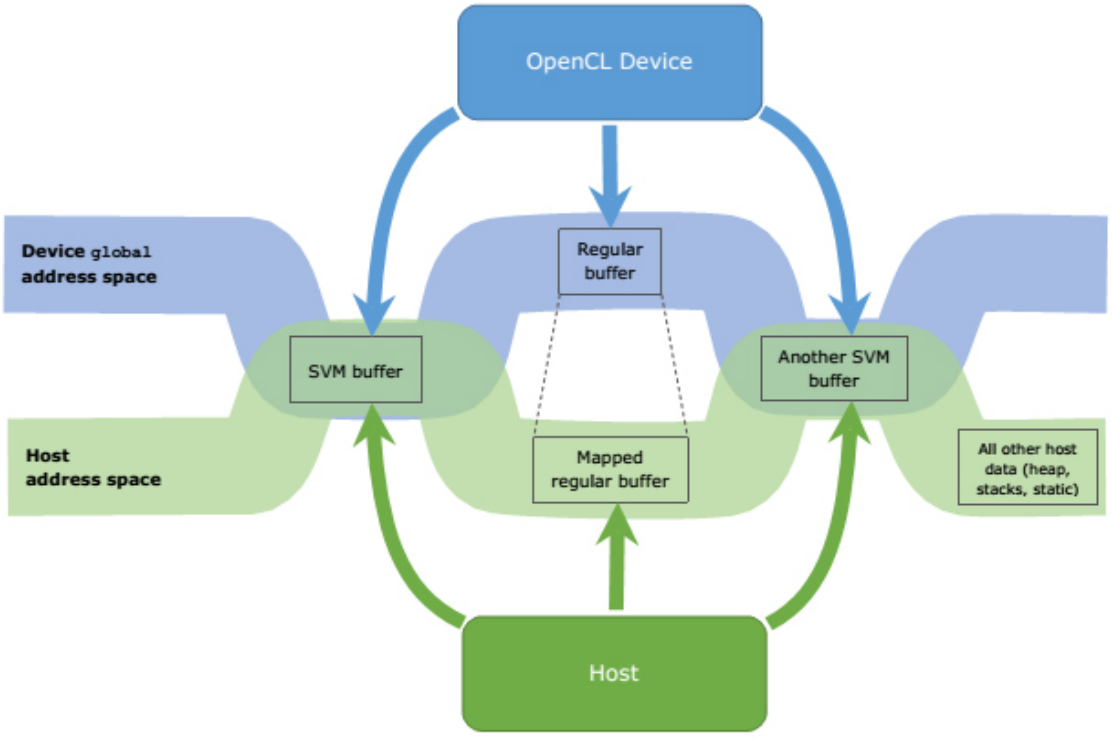
Standardowy sposób wykonania kernela OpenCL polega na kopiowaniu danych z pamięci hosta do pamięci globalnej lub stałej urządzenia w celu dalszego użycia. Wymaga to odpowiedniego przygotowania przez programistę buforów danych w pamięci gospodarza. Dodatkowo po stronie sprzętowej i programowej wymaga to dużej ilości operacji oraz stosowania wielu funkcji. Nawet dla architektur sprzętowo wspierających bezpośredni dostęp do pamięci RAM gospodarza (CellBE, APU), wymusza to podział pamięci na część dostępną tylko dla akceleratora oraz cześć dostępną dla gospodarza. Uniemożliwia to skuteczne przekazanie danych bez kopiowania (zero-copy) oraz zajmuje dodatkowy czas.

```
cl_mem cmPinnedBufIn = NULL; // input buffer mapped to host input buffer
cmPinnedBufIn = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_ALLOC_HOST_PTR,
 el_data_in_bytes, NULL, NULL);
cl_mem cmDevBufIn = NULL; // device input buffer allocated in card's memory

cmDevBufIn = clCreateBuffer(context, CL_MEM_READ_ONLY, el_data_in_bytes, NULL,
NULL);
double* el_data_in = NULL; // host input buffer (standard array)
el_data_in = (double*)clEnqueueMapBuffer(command_queue, cmPinnedBufIn, CL_TRUE,
CL_MAP_WRITE, 0, el_data_in_bytes, 0, NULL, NULL, NULL);
retval |= clSetKernelArg(kernel, 3, sizeof(cl_mem), (void *) &cmDevBufIn);

cl_mem cmPinnedBufOut = NULL; // output buffer mapped to host output buffer

cmPinnedBufOut = clCreateBuffer(context, CL_MEM_WRITE_ONLY | CL_MEM_ALLOC_HOST_PT
R, el_data_out_bytes, NULL, NULL);
cl_mem cmDevBufOut = NULL; // device output buffer allocated in card's memory

cmDevBufOut = clCreateBuffer(context, CL_MEM_WRITE_ONLY, el_data_out_bytes, NULL,
 NULL);
double* el_data_out = NULL; // host output buffer (standard array)
el_data_out = (double*)clEnqueueMapBuffer(command_queue, cmPinnedBufOut, CL_TRUE,
 CL_MAP_READ, 0, el_data_out_bytes, 0, NULL, NULL, NULL);
retval |= clSetKernelArg(kernel, 4, sizeof(cl_mem), (void *) &cmDevBufOut);
...
el_data_in[coeff] = ...
...
// writing input data to global GPU memory
clEnqueueWriteBuffer(command_queue, cmDevBufIn, CL_FALSE, 0, el_data_in_bytes, el
_data_in, 0, NULL, NULL);
...
// executing kernel
retval = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, globalWorkSize,
localWorkSize, 0, NULL,  &ndrEvt);
...
memset(el_data_out, 0, el_data_out_bytes);
// transfer output data
clEnqueueReadBuffer(command_queue, cmDevBufOut, CL_TRUE, 0, el_data_out_bytes,
el_data_out, 0, NULL, NULL);
```

```
double* el_data_in = (double*)clSVMAlloc(context,CL_MEM_READ_ONLY|CL_MEM_SVM_FINE_
GRAIN_BUFFER,el_data_in_bytes,NULL);

retval |= clSetKernelArgSVMPointer(kernel, 3, el_data_in);

double* el_data_out = (double*) clSVMAlloc(context,CL_MEM_WRITE_ONLY|CL_MEM_SVM_FI
NE_GRAIN_BUFFER,el_data_out_bytes,NULL);

retval |= clSetKernelArgSVMPointer(kernel, 4, el_data_out);
...
el_data_in[coeff] = ...
...

// executing kernel
retval = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, globalWorkSize, lo
calWorkSize, 0, NULL,  &ndrEvt);
```

Rysunek 4.8: Porównanie implementacji OpenCL 1.2 (lewo) z 2.0 (prawo)

Problemy te zostały rozwiązane wraz z wprowadzeniem wersji 2.0 standardu OpenCL w 2014 roku. Pozwala ona korzystać ze współdzielonej pamięci wirtualnej (Shared Virtual Memory), która unifikuje dla programisty całą dostępną pamięć. Stosowanie SVM ułatwia programowanie akceleratorów, znosząc obowiązek żmudnego przygotowywania danych oraz mapowania i przekazywania wskaźników do odpowiednich struktur. Wpływa to także na długość i czytelność kodu, co zostało pokazane na rysunku 4.8. Jak widać, klasyczna implementacja wymaga deklaracji dwóch osobnych buforów na dane - jednego po stronie gospodarza i jednego po stronie akceleratora, zarówno dla danych wejściowych (linijki 1-11), jak i wyjściowych (linijki 13-24). Dodatkowo poza skomplikowanym systemem mapowania adresów z jednego bufora na drugi (linijki 9 i 22), potrzebne są dodatkowe instrukcje przesyłu danych do (linijki 29-30) i z (linijki 38-39) akceleratora. W przypadku modelu SVM kod ulega znaczącemu skróceniu, jak widać na rysunku 4.8 po prawej stronie. SVM wymaga alokacji tylko po jednym buforze dla danych wejściowych (linijki 8-9) oraz wyjściowych (linijka 21-22) i nie wymaga osobnych procedur przesyłu danych przed ani po wykonaniu kernela (linijki 33-34).



Rysunek 4.9: Współdzielona pamięć wirtualna w OpenCL 2.0 [47]

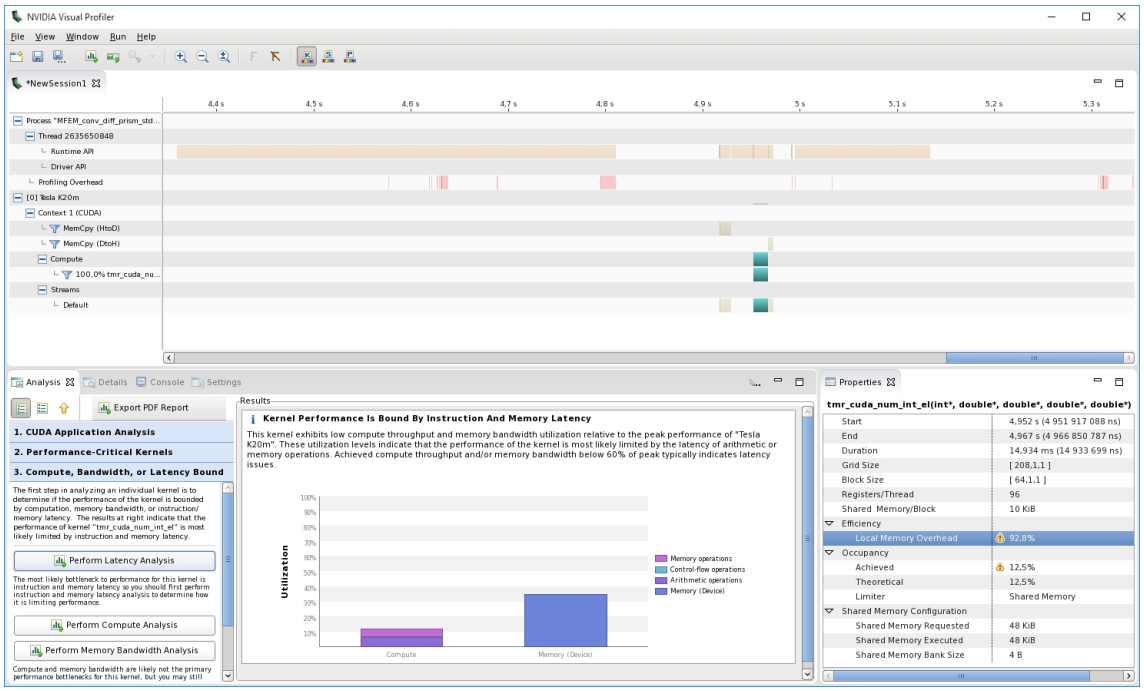
Z punktu widzenia wydajności, na architekturach zgodnych z modelem Heterogenous System Architecture (opisanych w rozdziale 2.5), użycie SVM pozwala w pełni wykorzystać potencjał heterogenicznej zunifikowanej pamięci (hUMA).

Oznacza to brak operacji kopiowania pomiędzy pamięcią gospodarza a akceleratorem, a co za tym idzie, znaczącą redukcję całkowitego czasu wykonania programu. W przypadku uruchomienia na architekturze niezgodnej z HSA, OpenCL sam zadba o najbardziej optymalne wykonanie i odpowiedni przesył danych. Współdzielona pamięć wirtualna ma postać bufora, który może być używany bezpośrednio zarówno przez akcelerator jak i CPU gospodarza (Rys 4.9).

Dzięki technologii OpenCL możliwym stało się pisanie programów które można uruchamiać na wielu typach akceleratorów. Jednakże nie rozwiązało to problemu zachowania wydajności wraz z przenośnością, gdyż każda z architektur wymaga osobnych optymalizacji. Problem ten został rozwiązany w niniejszej pracy poprzez stworzenie systemu automatycznego tuningu badanego algorytmu, opisanego w dalszych rozdziałach.

4.1.4. Analiza wykonania

Do analizy otrzymanego kodu używano, opracowanych przez autora, funkcji wbudowanych w źródła oraz narzędzi dostarczanych przez dostawców oprogramowania i sprzętu.

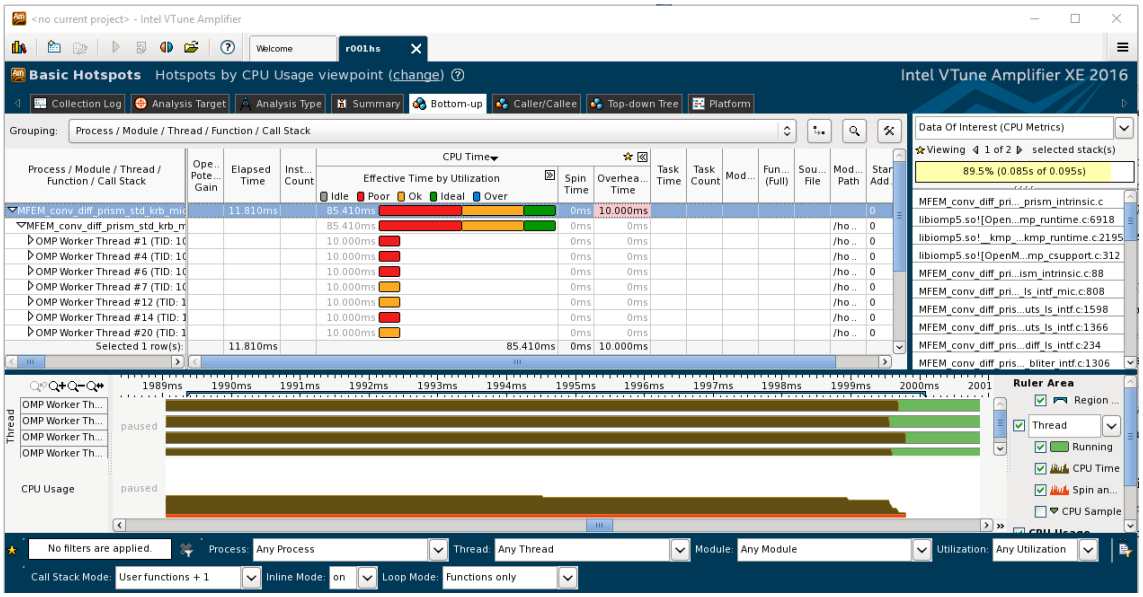


Rysunek 4.10: Nvidia Visual Profiler

W celu analizy kodu maszynowego (lub pseudo–maszynowego w przypadku zamkniętych rozwiązań firmy Nvidia) opracowano system liczenia liczby wystąpień określonych instrukcji dla najpopularniejszych z badanych architektur (Intel, Nvidia oraz

AMD). Pozwala on na określenie, jakie optymalizacje zostały użyte podczas kompilacji kodu oraz generuje podstawowe raporty dotyczące dostępów do pamięci i operacji arytmetycznych.

Dodatkowym narzędziem używanym do profilowania kodu był Nvidia Visual Profiler (Rys 4.10). Działa on w środowisku CUDA i dokonuje dynamicznej analizy programu w trakcie jego wykonania. Aplikacja ta wizualizuje przepływ instrukcji, co pozwala na znalezienie obszarów wymagających optymalizacji. Generuje także raporty mające sugerować zmiany, które mogą mieć wpływ na wydajność programu. Dzięki bezpośredniemu oparciu na sprzęcie, Nvidia Visual Profiler pozwala zmierzyć efektywność aplikacji bezpośrednio z liczników sprzętowych, co dla zamkniętych rozwiązań firmy Nvidia jest jedyną metodą na głębszą analizę wykonania [95].



Rysunek 4.11: Intel Vtune Amplifier

Komplementarnym narzędziem dla akceleratorów firmy AMD jest CodeXL pozwalający na debugging oraz analizę kodu w środowisku OpenCL. Jest to rozwiązanie otwarte i pozwalające na analizę wykonania zarówno na akceleratorach opartych na GPU jak i na zwykłych CPU oraz jednostkach hybrydowych APU.

Podczas analizy wywołania na procesorach ogólnego przeznaczenia firmy Intel oraz koprocesorze Intel Xeon Phi, używano profilera Intel Vtune Amplifier dostępnego w pakiecie Intel Parallel Studio XE (Rys. 4.11). Narzędzie to pozwala na głęboką analizę kodu, z dokładnym podglądem wielowątkowości oraz użyciem liczników sprzętowych. Pozwala to zidentyfikować różne problemy związane z wydajnością programu oraz bardziej dokładnie wykorzystać dostępne zasoby sprzętowe.

4.2. Podsumowanie

Wykorzystane narzędzia i języki programowania pozwoliły na szczegółową implementację oraz analizę badanego algorytmu. Dzięki dokładnej analizie kodu oraz profilowaniu uruchomienia algorytmu na różnych architekturach sprzętowych, autor poszukiwał głównych elementów badanej procedury, mających wpływ na wydajność. Równocześnie dzięki przeprowadzonym analizom, autor próbował stwierdzić, w jakim stopniu poszczególne elementy budowy danego sprzętu wpływają na wykonanie badanego algorytmu całkowania numerycznego.

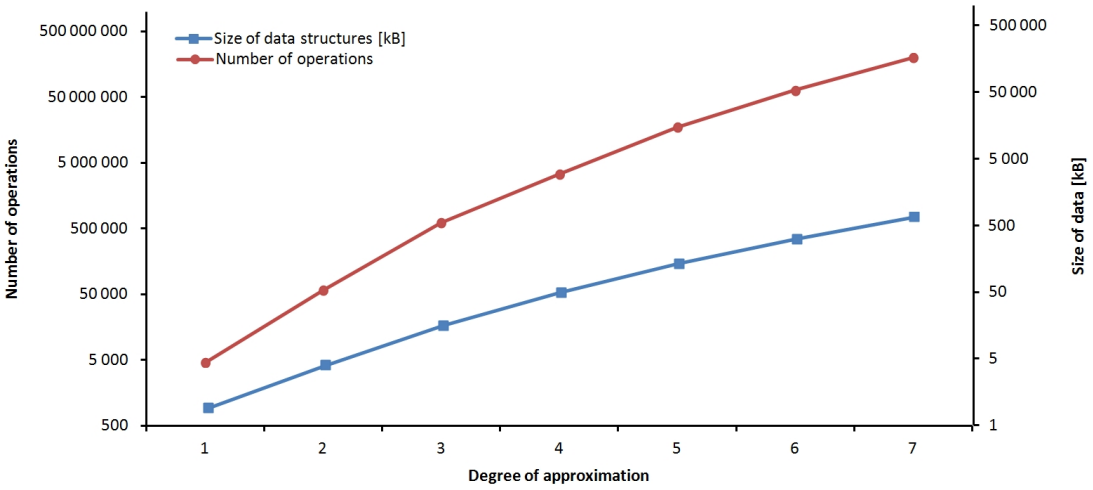
Rozdział 5

Implementacja i wyniki badań

5.1. PowerXCell 8i

5.1.1. Metodologia

Jak wspomniano we wcześniejszych rozdziałach, procesor PowerXCell 8i został użyty w zadaniu Poissona przy zastosowaniu nieciągłej aproksymacji Galerkina. W związku z tym, że dla każdej z jednostek obliczeniowych SPE dostępne jest 256 kB pamięci, oraz na podstawie danych z tabeli 3.5 pomnożonej przez rozmiar używanego typu danych (*float* – 4 bajty), można wywnioskować, że dla stopni aproksymacji niższych niż 5 dane całkowania dotyczące funkcji kształtu oraz ich pochodnych dla wszystkich punktów Gaussa, powinny zmieścić się w pamięci lokalnej SPE. Dla aproksymacji wyższych rzędów, wartości te nie tylko nie zmieszczą się w pamięci, ale również wynikowa macierz sztywności będzie zbyt duża do przechowania w jednej tablicy. W związku z tym faktem, zmodyfikowano oryginalny algorytm, stosując podział macierzy sztywności na mniejsze części i wykonując obliczenia dla każdej z nich osobno. Pętla po fragmentach macierzy sztywności występuje przed pętlą po punktach Gaussa (linijka 8 w Algorytmie 1). Oznacza to, że obliczenia dotyczące transformacji jakobianowych, pochodnych funkcji kształtu oraz współczynników całkowania (linijki 9 – 11 w Algorytmie 1) są powtarzane tyle razy, na ile części podzielono macierz sztywności. Z tego powodu można wysnuć wniosek, że mały rozmiar lokalnej pamięci (Local Store) powoduje wzrost ilości obliczeń dla aproksymacji wyższych rzędów, a co za tym idzie wzrost ilości czasu potrzebnego na wykonanie algorytmu. Jak widać z rysunku 5.1, liczba operacji rośnie szybciej niż rozmiar danych wraz ze wzrostem stopnia aproksymacji. Pozwala to wysnuć wniosek, iż ogólna wydajność algorytmu powinna zależeć coraz bardziej od efektywnego wykonania wewnętrznych pętli po funkcjach kształtu. Wydajność tych pętli w dużej mierze zależy od dokładnego zmapowania obliczeń na architekturę. W tym celu, należy skorzystać z dwóch mechanizmów charakterystycznych dla SPE - dwóch potoków



Rysunek 5.1: Rozmiar danych oraz liczba operacji jako funkcja stopnia aproksymacji

przetwarzania oraz jednostek wektorowych operujących na 128 - bitowych rejestrach. W celu wykorzystania jednostek SIMD użyto typu *float4*, wbudowanego w język OpenCL. W związku z przedstawionymi powyżej założeniami skonstruowano wersję algorytmu całkowania numerycznego w oparciu o dekompozycję danych. Polega ona na podziale macierzy sztywności pomiędzy wątki. Algorytm 8, oznaczony jako DD (*data decomposition*), posiada dwie wersje zależne od sposobu uzyskiwania funkcji kształtu i ich pochodnych. Pierwsza wersja, oznaczona DD_SR, sczytuje je z lokalnych tablic i działa jedynie dla stopni aproksymacji niższych niż 5. Druga wersja (DD_SC), dokonuje obliczeń wartości funkcji kształtu oraz ich pochodnych w każdym SPE.

W przypadku zadania Poissona, można rozważać jeszcze inną opcję algorytmu całkowania numerycznego. W związku z tym, że zadanie to posiada jeszcze pętlę po wymiarach przestrzeni i_D, możemy ją zwektoryzować celem wykonania równoległego. Zamiast standardowego sumowania wszystkich składowych ze współrzędnych x, y, z dla każdego punktu całkowania, możemy dokonać osobnych obliczeń dla każdego wymiaru. Dzięki temu, operację sumowania wszystkich składowych, dokonujemy tuż przed zapisem części macierzy sztywności do pamięci globalnej. Tą wersję algorytmu oznaczamy WF - zrównoleglenie oparte na sformułowaniu słabym (*weak formulation*). Algorytm ten również posiada dwie wersje: WF_SR i WF_SC, w zależności od sposobu uzyskiwania funkcji kształtu i ich pochodnych. Wadą tego podejścia, może być zwiększone zapotrzebowanie na pamięć, w związku z osobnym przechowywaniem danych dla każdego wymiaru przestrzeni. Algorytm 9 przedstawia wersję WF procedury całkowania numerycznego na procesorze PowerXCell.

Algorytm 8: Algorytm całkowania numerycznego z dekompozycją danych na procesor PowerXCell (wersja DD).

- wczytaj z globalnej pamięci tablice $\boldsymbol{\xi}^Q$ i $\boldsymbol{w}^Q$ z danymi całkowania numerycznego dla wszystkich punktów całkowania i zapisz w lokalnych tablicach;
- *(wariant DD_SR)* wczytaj wartości wszystkich funkcji kształtu oraz ich pochodnych względem współrzędnych lokalnych we wszystkich punktach całkowania w elemencie odniesienia;

for $e = 1$ *to* N_{EL_SPE} **do**

- wczytaj potrzebne dane dotyczące geometrii elementu z pamięci globalnej do lokalnych tablic;

for $ipart = 1$ *to* $N_{czesci\_macierzy\_sztywnosci}$ **do**

- zainicjuj elementową macierz sztywności $\boldsymbol{A}^e$ w pamięci lokalnej;

for $i_Q = 1$ *to* N_Q **do**

- sczytaj z lokalnych tablic *(wariant DD_SR)* lub oblicz *(wariant DD_SC)* wartości funkcji kształtu i ich pochodnych względem współrzędnych lokalnych w elemencie odniesienia;
- oblicz potrzebne dane transformacji jakobianowych ($\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\xi}}$, $\frac{\partial \boldsymbol{\xi}}{\partial \boldsymbol{x}}$, **vol**);
- korzystając z macierzy jakobianowych oblicz pochodne funkcji kształtu względem współrzędnych globalnych – $\boldsymbol{\phi}[i_Q]$ i zapisz je jako cztery tablice wektorów typu *float4* (jeden wektor dla czterech kolejnych pochodnych funkcji kształtu);

for $i_S = 1$ *to* N_S **do**

for $j_S = 1$ *to* $N_S/4$ **do**

- oblicz 4 kolejne wyrazy macierzy sztywności związane z indeksem j_S przechowywane w wektorze *float4* ($\boldsymbol{A}^e_4[j_S]$), przy użyciu rejestrów oraz lokalnych tablic wektorów *float4* z funkcjami kształtu i pochodnymi zgodnie z wzorem:

$$\boldsymbol{A}^e_x[j_S] = \left(\frac{\partial \boldsymbol{\psi}[i_S]}{\partial x} \cdot \frac{\partial \boldsymbol{\psi}[j_S]_x}{\partial x} + \frac{\partial \boldsymbol{\psi}[i_S]}{\partial y} \cdot \frac{\partial \boldsymbol{\psi}[j_S]_x}{\partial y} + \frac{\partial \boldsymbol{\psi}[i_S]}{\partial z} \cdot \frac{\partial \boldsymbol{\psi}[j_S]_x}{\partial z}\right) \cdot \mathbf{vol}^Q[i_Q]$$

$$\boldsymbol{A}^e_y[j_S] = \left(\frac{\partial \boldsymbol{\psi}[i_S]}{\partial x} \cdot \frac{\partial \boldsymbol{\psi}[j_S]_y}{\partial x} + \frac{\partial \boldsymbol{\psi}[i_S]}{\partial y} \cdot \frac{\partial \boldsymbol{\psi}[j_S]_y}{\partial y} + \frac{\partial \boldsymbol{\psi}[i_S]}{\partial z} \cdot \frac{\partial \boldsymbol{\psi}[j_S]_y}{\partial z}\right) \cdot \mathbf{vol}^Q[i_Q]$$

$$\boldsymbol{A}^e_z[j_S] = \left(\frac{\partial \boldsymbol{\psi}[i_S]}{\partial x} \cdot \frac{\partial \boldsymbol{\psi}[j_S]_z}{\partial x} + \frac{\partial \boldsymbol{\psi}[i_S]}{\partial y} \cdot \frac{\partial \boldsymbol{\psi}[j_S]_z}{\partial y} + \frac{\partial \boldsymbol{\psi}[i_S]}{\partial z} \cdot \frac{\partial \boldsymbol{\psi}[j_S]_z}{\partial z}\right) \cdot \mathbf{vol}^Q[i_Q]$$

$$\boldsymbol{A}^e_v[j_S] = \left(\frac{\partial \boldsymbol{\psi}[i_S]}{\partial x} \cdot \frac{\partial \boldsymbol{\psi}[j_S]_v}{\partial x} + \frac{\partial \boldsymbol{\psi}[i_S]}{\partial y} \cdot \frac{\partial \boldsymbol{\psi}[j_S]_v}{\partial y} + \frac{\partial \boldsymbol{\psi}[i_S]}{\partial z} \cdot \frac{\partial \boldsymbol{\psi}[j_S]_v}{\partial z}\right) \cdot \mathbf{vol}^Q[i_Q]$$

, gdzie x, y, z, v oznaczają kolejne funkcje kształtu związane z indeksem j_S;

end

end

end

- zapisz fragment macierzy sztywności w pamięci globalnej;

end

end

Algorytm 9: Algorytm całkowania numerycznego na procesor PowerXCell z wektoryzacją opartą na sformułowaniu słabym (wersja WF).

- wczytaj z globalnej pamięci tablice $\boldsymbol{\xi}^Q$ i $\boldsymbol{w}^Q$ z danymi całkowania numerycznego dla wszystkich punktów całkowania i zapisz w lokalnych tablicach;
- *(Wariant WF_SR)* wczytaj wartości wszystkich funkcji kształtu oraz ich pochodnych względem współrzędnych lokalnych we wszystkich punktach całkowania w elemencie odniesienia;
- **for** $e = 1$ *to* N_{EL_SPE} **do**
  - wczytaj potrzebne dane dotyczące geometrii elementu z pamięci globalnej do lokalnych tablic;
  - **for** $ipart = 1$ *to* $N_{czesci\_macierzy\_sztywnosci}$ **do**
    - zainicjuj fragment elementowej macierzy sztywności $\boldsymbol{A}^e$ w pamięci lokalnej;
    - **for** $i_Q = 1$ *to* N_Q **do**
      - sczytaj z lokalnych tablic *(wariant WF_SR)* lub oblicz *(wariant WF_SC)* wartości funkcji kształtu i ich pochodnych względem współrzędnych lokalnych w elemencie odniesienia;
      - oblicz potrzebne dane transformacji jakobianowych ($\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\xi}}$, $\frac{\partial \boldsymbol{\xi}}{\partial \boldsymbol{x}}$, **vol**);
      - oblicz pochodne funkcji kształtu względem współrzędnych globalnych – $\boldsymbol{\phi}[i_Q]$ i zapisz je w tablicy wektorów typu *float4* (jeden wektor na trzy pochodne);
      - **for** $i_S = 1$ *to* N_S **do**
        - **for** $j_S = 1$ *to* N_S **do**
          - uaktualnij wektor *float4* $\boldsymbol{A}^e_4$ przechowujący trzy części macierzy $\boldsymbol{A}^e_{ij}$ skojarzone z każdym z wymiarów przestrzeni x, y, z zgodnie ze wzorem (3.7):
          - $\boldsymbol{A}^e_x[i_S][j_S] = \frac{\partial \boldsymbol{\psi}[i_S]}{\partial x} \cdot \frac{\partial \boldsymbol{\psi}[j_S]}{\partial x} \cdot \mathbf{vol}^Q[i_Q]$
          - $\boldsymbol{A}^e_y[i_S][j_S] = \frac{\partial \boldsymbol{\psi}[i_S]}{\partial y} \cdot \frac{\partial \boldsymbol{\psi}[j_S]}{\partial y} \cdot \mathbf{vol}^Q[i_Q]$
          - $\boldsymbol{A}^e_z[i_S][j_S] = \frac{\partial \boldsymbol{\psi}[i_S]}{\partial z} \cdot \frac{\partial \boldsymbol{\psi}[j_S]}{\partial z} \cdot \mathbf{vol}^Q[i_Q]$
        - **end**
      - **end**
    - **end**
    - zsumuj trzy elementy wektora $\boldsymbol{A}^e_x + \boldsymbol{A}^e_y + \boldsymbol{A}^e_z$ dla każdego elementu macierzy A^e_{ij} i zapisz fragment macierzy sztywności do pamięci globalnej;
  - **end**
- **end**

Tabela 5.1: Charakterystyki wykonania wersji WF algorytmu całkowania numerycznego na procesorze PowerXCell dla problemu Poissona i różnych stopni aproksymacji p.

	Stopień aproksymacji p						
	1	*2*	*3*	*4*	*5*	*6*	*7*
Rozmiar dopasowanej macierzy sztywności [kB]	0,19	1,73	6,40	24	64,50	157	332
Ilość fragmentów macierzy sztywności	1	1	1	1	2	5	9
Ilość wierszy w każdym fragmencie macierzy sztywności	8	24	40	80	64	40	32
Rozmiar fragmentu macierzy sztywności [kB]	0,77	6,91	25,60	97,28	129,00	125,40	147,40
Rozmiar tablicy na dane funkcji kształtu [kB]	0,18	0,43	0,85	1,50	2,41	3,64	5,23
Liczba operacji (w milionach)	0,004	0,072	0,657	3,965	20	76,42	238

Ostatnim krokiem, w poprawnym zmapowaniu algorytmu na procesor, jest wykorzystanie w pełni jego możliwości przetwarzania potokowego. Jako że nie istnieją mechanizmy, pozwalające na bezpośrednią kontrolę nad przepływem instrukcji, osiągnięcie tego jest zależne od odpowiednich optymalizacji kodu. Rezultat tych zabiegów został sprawdzony za pomocą narzędzia *spu_timing* dostarczanego wraz z pakietem programistycznym (SDK) procesorów Cell. Narzędzie to pozwala na analizę kodu assemblera poprzez prezentację operacji wykonywanych przez SPE w każdym takcie zegara.

Główne optymalizacje kodu algorytmu dotyczą ręcznego rozwinięcia pętli po funkcjach kształtu N_S. Dla wersji WF możliwe stało się rozwinięcie zewnętrznej pętli ze współczynnikiem 2, a wewnętrznej z współczynnikiem równym 8. Dla wersji DD dokonano rozwinięcia obu pętli ze współczynnikiem wynoszącym 2.

W przypadku implementacji w języku OpenCL, kod hosta odpowiedzialny jest za obliczenie rozmiarów oraz przygotowanie danych używanych przez SPE do obliczeń. Musi on również dokonać odpowiedniego podziału danych w zależności od dostępnych zasobów oraz zadbać o właściwe dopasowanie (padding) tablic. Obliczone charakterystyki wykonania dla algorytmu WF prezentuje tabela 5.1.

Dane w tabeli odpowiadają wykonaniu algorytmu WF z liczbą operacji dla wersji WF_SC. Liczba ta będzie przekraczać standardowo obliczaną liczbę operacji dla algorytmu. Wynika to z faktu, że tablice przechowujące fragmenty macierzy sztywności są dopasowane w pamięci, oraz z tego że 128 bitowe wektory przechowują jedynie trzy 32-bitowe wartości. W związku z tym ostatnim, należy także zauważyć, że każdy SPE przesyła zawartość swoich 128-bitowych rejestrów do/z pamięci lokalnej, gdy tymczasem tylko 96 bitów zawiera prawdziwe dane potrzebne do obliczeń. Im-

plikuje to fakt, że efektywna wydajność procesora jest niższa, niż ta liczona z liczby wykonanych operacji.

Dla wersji DD algorytmu uzyskano podobne charakterystyki wykonania z główną różnicą w ilości operacji, która osiąga wielkości zbliżone do tych z tabeli 5.1.

Dzięki budowie procesora PowerXCell i jego bezpośredniemu dostępowi do pamięci hosta, nie jest koniecznym dokonywanie czasochłonnych transferów danych, a jedynie ich przygotowanie w postaci odpowiednich buforów pamięci globalnej do których dostęp jest optymalizowany przez kompilator.

5.1.2. Wyniki

Opracowane algorytmy zostały przetestowane na serwerze IBM BladeCenter QS22/LS21 wyposażonym w węzły QS22 z dwoma procesorami IBM PowerXCell 8i o częstotliwości 3.2 GHz i 8GB pamięci RAM. W warstwie programistycznej używano frameworka IBM OpenCL SDK 0.3 z kompilatorem gcc. Dzięki użyciu dwóch procesorów PowerXCell, możliwym stało się skorzystanie z 16 jednostek SPE, przy jednym rdzeniu PPE odpowiadającym za proces hosta. Jako system referencyjny użyto procesora Intel Core i5–2520M wyposażonego w rdzeń Sandy Bridge opisywany w poprzednich rozdziałach. Uruchomiono na nim zoptymalizowany algorytm całkowania numerycznego z włączoną automatyczną wektoryzacją.

Tabela 5.2: Czas wykonania algorytmu całkowania numerycznego (w mikrosekundach) dla problemu modelowego, jednego elementu, różnych wariantów algorytmu oraz zmiennych stopni aproksymacji.

	Stopień aproksymacji p						
	1	*2*	*3*	*4*	*5*	*6*	*7*
Pojedynczy rdzeń Sandy Bridge	0,809	6,417	43	242	1070	3659	11505
PowerXCell, 16 SPE							
inicjalizacja	0,123	0,741	2,6	9,69	25,7	62	132
DD_SC – czas całkowity	0,697	4,841	21,65	75	228,9	773	2691
DD_SR – czas całkowity	0,546	3,271	17,10	63,59	–	–	–
WF_SC – czas całkowity	0,601	3,944	17,63	68,72	276,3	1064	3348
WF_SR – czas całkowity	0,458	2,755	13,95	63,32	–	–	–
PowerXCell, 1 SPE							
inicjalizacja	0,119	0,703	2,51	9,34	24,9	59	127
DD_SC – czas całkowity	8,685	61,83	291,2	992	3115	11077	40584
DD_SR – czas całkowity	6,265	47,16	217,8	815,1	–	–	–
WF_SC – czas całkowity	7,256	48,04	229,4	900,1	3893	15738	50987
WF_SR – czas całkowity	4,989	29,09	166,6	792,9	–	–	–

W celu ograniczenia wzrostu pamięci potrzebnej na przechowywanie macierzy sztywności, założono jej stały rozmiar, co w przypadku rosnącego stopnia aprok-

Tabela 5.3: Wydajność algorytmu całkowania numerycznego dla problemu modelowego, jednego elementu, różnych wariantów algorytmu oraz zmiennych stopni aproksymacji.

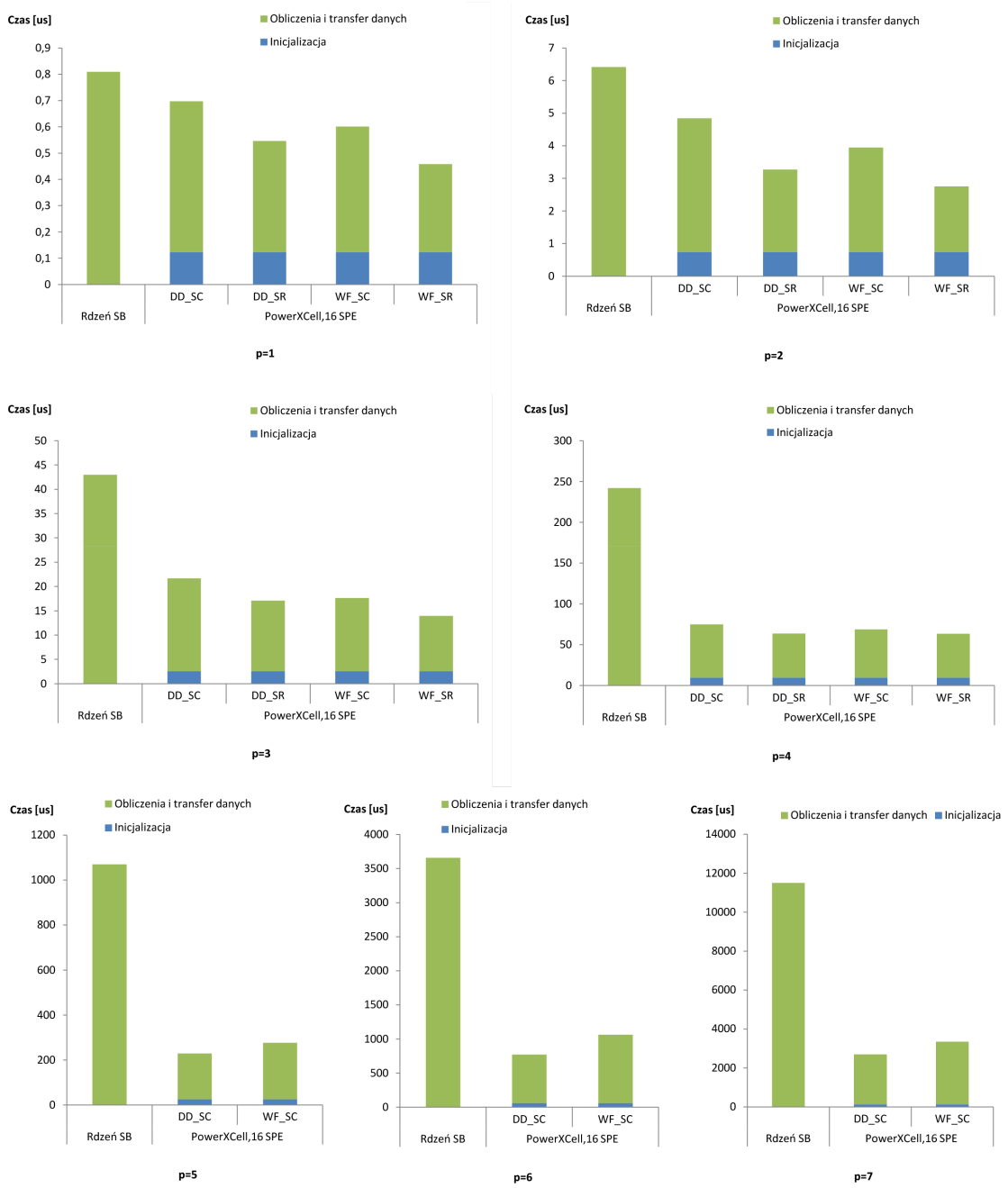
Wydajność w GFlops	**Stopień aproksymacji p**						
	1	*2*	*3*	*4*	*5*	*6*	*7*
Pojedynczy rdzeń SB	4,50	8,04	13,66	13,58	15,96	17,23	17,13
4-rdzenie Sandy Bridge	18,02	32,16	54,63	54,34	63,84	68,94	68,51
DD_SC – wewnętrzna	8,16	17,33	33,40	54,38	86,77	93,26	79,86
DD_SC – zewnętrzna	5,22	10,66	27,12	43,83	74,61	81,51	73,22
DD_SR – wewnętrzna	9,66	21,39	43,09	65,34	–	–	–
DD_SR – zewnętrzna	6,67	15,77	34,34	51,69	–	–	–
WF_SC – wewnętrzna	11,43	32,50	64,29	100,39	118,46	112,32	108,53
WF_SC – zewnętrzna	6,06	13,08	33,29	47,84	61,80	59,22	58,84
WF_SR – wewnętrzna	14,43	40,14	84,29	111,52	–	–	–
WF_SR – zewnętrzna	7,94	18,72	42,07	51,92	–	–	–

symacji oznacza mniejszą ilość elementów przetwarzanych przez kernel. W związku z powyższym zadanie w każdym przypadku zajmowało ok 500 MB pamięci globalnej DRAM. Konsekwencją tego jest fakt, że narzut obliczeń jest prawie zawsze taki sam niezależnie od wzrastającego p, oraz zostaje zamortyzowany poprzez zmniejszającą się ilość przetwarzanych elementów.

Wynik eksperymentów obrazuje tabela 5.2. Pokazuje ona czas wykonania dla pojedynczego elementu, powstały z podzielenia czasu całkowitego przez liczbę przetwarzanych elementów. Czasy zostały podzielone ze względu na użyty wariant algorytmu oraz dodatkowo na czas inicjalizacji oraz czas całkowity.

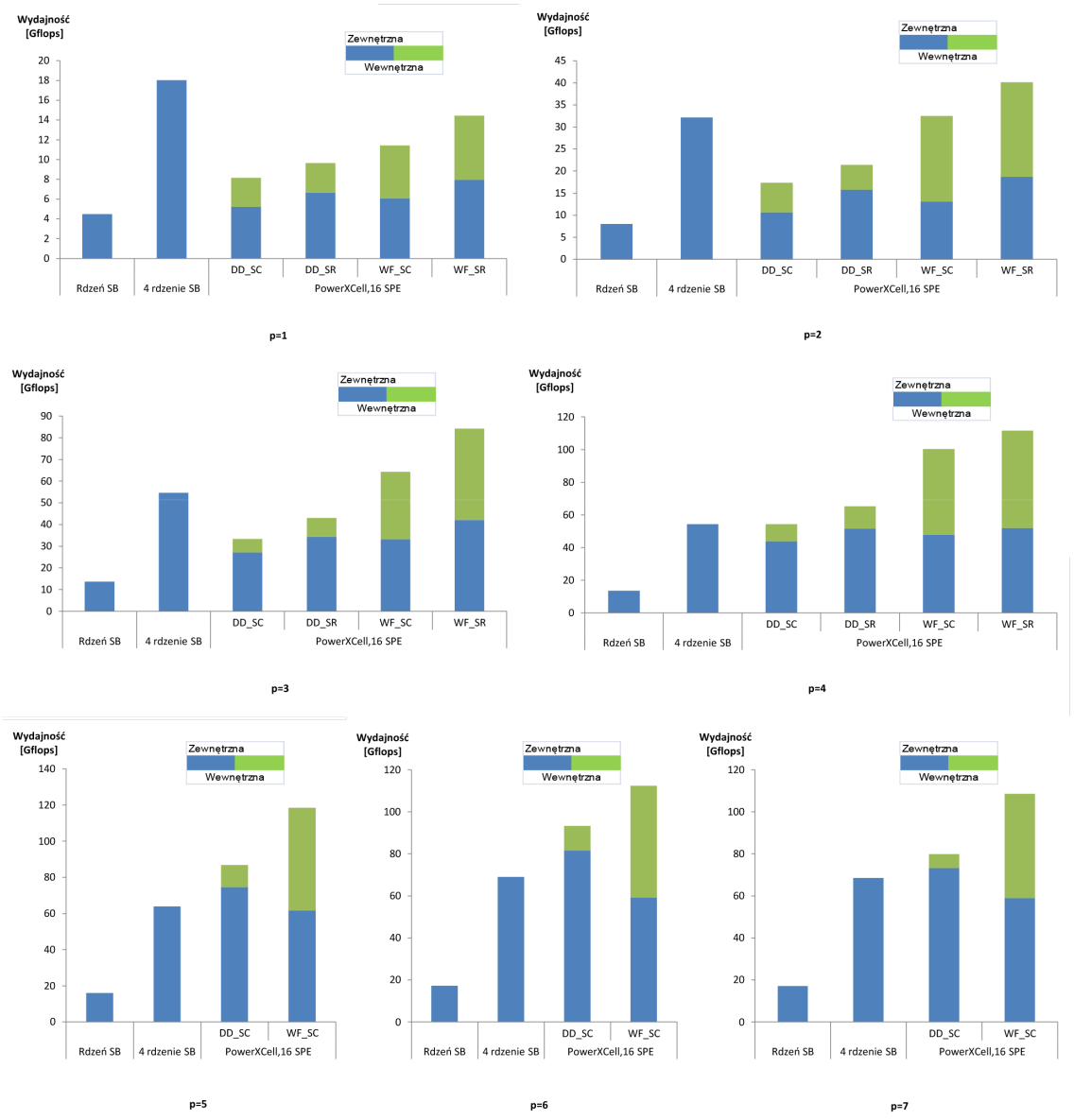
Wyniki te zaprezentowano na rysunku 5.2. Z wykresów można zauważyć, że korzyść ze stosowania wektorowych jednostek SPE jest zauważalna dopiero przy wyższych stopniach aproksymacji. Związane to jest ze zwiększającym się stosunkiem ilości obliczeń do ilości potrzebnych transferów danych z i do pamięci.

Tabela 5.3 przedstawia charakterystyki wykonania obliczone na podstawie zmierzonych wartości. Zmierzona wydajność pojedynczego rdzenia Sandy Bridge została ekstrapolowana na typową ilość czterech rdzeni z założeniem perfekcyjnej skalowalności. Dla każdej z wersji algorytmu uruchamianego na procesorze PowerXCell zostały zaprezentowane dwa wyniki. Pierwszym jest wydajność obserwowana z perspektywy zewnętrznej – liczba operacji używanych w algorytmie (z tabeli 5.1) jest dzielona przez całkowity czas wykonania. Druga wartość (wewnętrzna) została obliczona dzieląc liczbę rzeczywiście wykonanych operacji (z analizy asemblera) przez czas wykonania samego kernela. W przypadku wersji WF algorytmu, różnice pomiędzy tymi wartościami są związane nie tylko z fazą inicjalizacji, ale również z faktem wykonywania przez procesor ponadmiarowych operacji (tabela 5.1).



Rysunek 5.2: Czas wykonania algorytmu całkowania numerycznego(w mikrosekundach) dla problemu modelowego, jednego elementu, różnych wariantów algorytmu oraz zmiennych stopni aproksymacji.

Rysunek 5.3 przedstawia wyniki wydajnościowe z tabeli 5.3. Z wykresów można zauważyć, że rdzenie Sandy Bridge oferują podobną wydajność jak PowerXCell 8i, mimo kilkuletniej różnicy w architekturze tych procesorów. Świadczy to o bardzo dobrym wykorzystaniu wektoryzacji przez nowoczesne architektury rdzeni ogólnego przeznaczenia. Dla rdzeni PowerXCell wyniki można uznać za satysfakcjonujące dla wyższych stopni aproksymacji. Podobnie do rdzeni Sandy Bridge, jednostki SPE



Rysunek 5.3: Wydajność algorytmu całkowania numerycznego dla problemu modelowego, jednego elementu, różnych wariantów algorytmu oraz zmiennych stopni aproksymacji.

potrafią dobrze zutylizować rosnący współczynnik ilości obliczeń w zwektoryzowanych pętlach do obliczeń poza nimi. Mimo dużej ilości transferów danych związanych z ograniczoną ilością pamięci lokalnej i rejestrów, oraz istnienia fragmentów kodu gdzie korzystanie z operacji FMA jest niemożliwe, wydajność dla wersji DD_SC osiągnęła 79 GFlops (19% teoretycznego maksimum) a dla wersji WF_SC - 108 GFlops (26% teoretycznego maksimum). Dla niższych stopni aproksymacji, narzut związany z inicjalizacją oraz transferami danych z i do pamięci globalnej, znacząco obniża wydajność algorytmu. W tym przypadku, także liczba operacji w optymalizowanych pętlach dla pojedynczego elementu macierzy sztywności jest zbyt mała w stosunku do liczby operacji poza nimi.

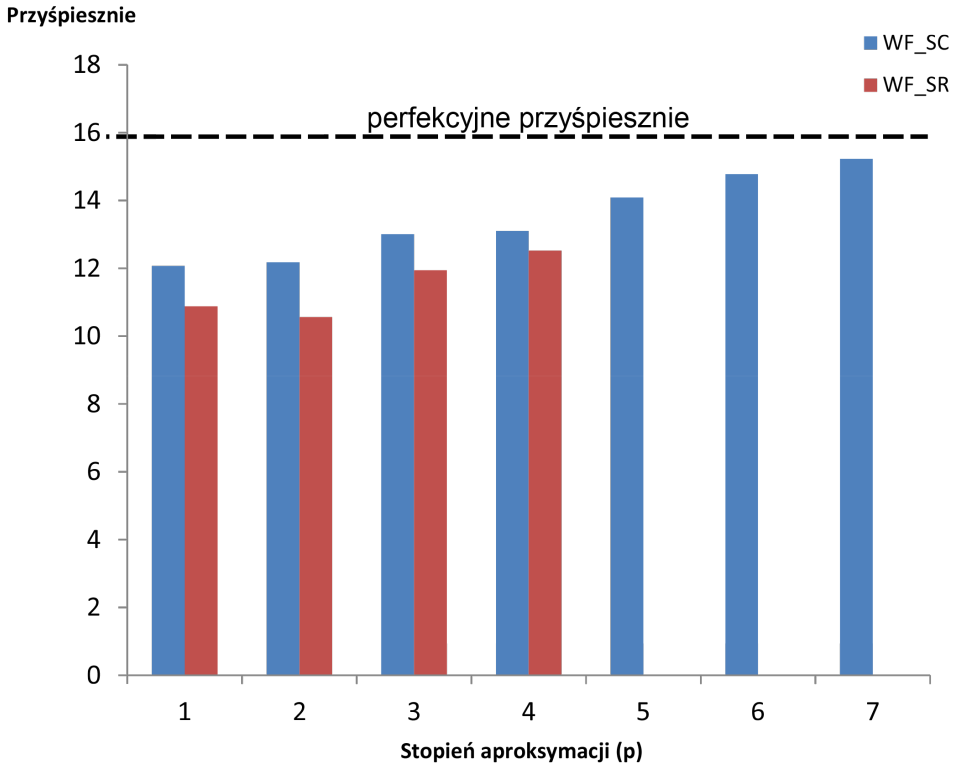
Analiza kodu assemblera pokazuje, że w przypadku optymalizowanej pętli po funkcjach kształtu, wydajność jest zbliżona do optymalnej, poprzez wykorzystanie obu potoków przetwarzania pracujących równocześnie bez przestojów (Rys. 5.4).

```
007681 0D                    123456          fma     $118,$124,$67,$118
007681 1D                    123456          lqd     $79,160($53)
007682 0D                     234567         fma     $120,$124,$65,$120
007682 1D                     234567         lqd     $123,176($53)
007683 0D                      345678        fma     $67,$124,$67,$68
007683 1D                      345678        lqd     $121,192($53)
007684 0D                       456789       fma     $68,$124,$69,$70
007684 1D                       456789       lqd     $119,208($53)
007685 0D                        567890      fma     $65,$124,$65,$66
007685 1D                        567890      lqd     $117,224($53)
007686 0D                         678901     fma     $69,$124,$69,$72
007686 1D                         678901     lqd     $66,240($53)
007687 0D                          789012    fma     $70,$124,$71,$74
007687 1D                          789012    stqd    $118,16($53)
007688 0D                           890123   fma     $71,$124,$71,$75
007688 1D                           890123   stqd    $120,0($53)
007689 0D                            901234  fm      $78,$62,$122
007689 1D                            901234  stqd    $67,48($53)
007690 0D                             012345 fm      $0,$63,$122
007690 1D                             012345 stqd    $68,64($53)
007691 0D                              123456fm      $122,$64,$122
007691 1D                              123456stqd    $65,32($53)
007692 0D                               234567fma    $72,$124,$73,$76
007692 1D                               234567stqd    $69,80($53)
007693 0D                                345678fma    $65,$124,$73,$77
007693 1D                                345678stqd    $70,96($53)
007694 0D                                 45    ai      $56,$56,32
007694 1D                                 456789stqd    $71,112($53)
```

Rysunek 5.4: Pełne wykorzystanie dwóch potoków wykonania w algorytmie całkowania numerycznego na procesorze PowerXCell 8i

Na rysunku widać, że w tym samym takcie zegara dokonywana jest operacja fma oraz odczyt/zapis pamięci. Widać także oznaczenia poszczególnych potoków - 0D oraz 1D. Czynnikiem ograniczającym to wykonanie jest liczba dostępów do pamięci dla 128 bitowych rejestrów, która jest większa niż liczba operacji SIMD wewnątrz pętli po funkcjach kształtu.

Kolejną charakterystyką algorytmu całkowania numerycznego jest stopnień przyśpieszenia w porównaniu do wykonania sekwencyjnego (na pojedynczym rdzeniu). Z wykresu 5.5 można zauważyć, że współczynnik przyśpieszenia dla 16 rdzeni zmie-



Rysunek 5.5: Przyśpieszenie osiągnięte przez wersję WF algorytmu całkowania numerycznego dla problemu modelowego i różnych stopni aproksymacji.

nia się w zakresie od ok. 11 do prawie 16. Daje to wydajność zrównoleglenia zawsze powyżej 68%, a dla stopnia aproksymacji $p = 7$ osiąga ponad 95%.

5.1.3. Wnioski

Porównując różne wersje badanego algorytmu można wywnioskować, że wersje z odczytem wartości funkcji kształtu i ich pochodnych z pamięci (SR) dają zawsze lepsze wyniki wydajnościowe, zaś wersje SC (z funkcjami kształtu obliczanymi lokalnie) wykazują większą elastyczność, poprzez swoje mniejsze wymagania dotyczące pamięci.

Wybór pomiędzy wariantem DD oraz WF jest zależny od problemu. Dla badanego przez autora problemu Poissona, oba podejścia wydają się w miarę równoważne. Jednakże dla sytuacji w której np. dokonujemy dekompozycji danych w celu podziału pomiędzy rdzenie lub węzły różnych typów, dodatkowa optymalizacja oparta na sformułowaniu słabym może przynieść znaczące korzyści.

Uzyskane wyniki wydajnościowe, świadczą o dużym potencjale zarówno sprzętu jak i algorytmu w kwestii wzajemnego dopasowania. Procesor PowerXCell będący swoistym przodkiem rozwiązań stosowanych obecnie (APU, Xeon Phi) okazał

się być przydatnym narzędziem do badania możliwości współczesnych architektur bazujących na wielopoziomowej hierarchii pamięci oraz jednostkach wektorowych.

5.2. Procesory ogólnego przeznaczenia (CPU)

5.2.1. Metodologia

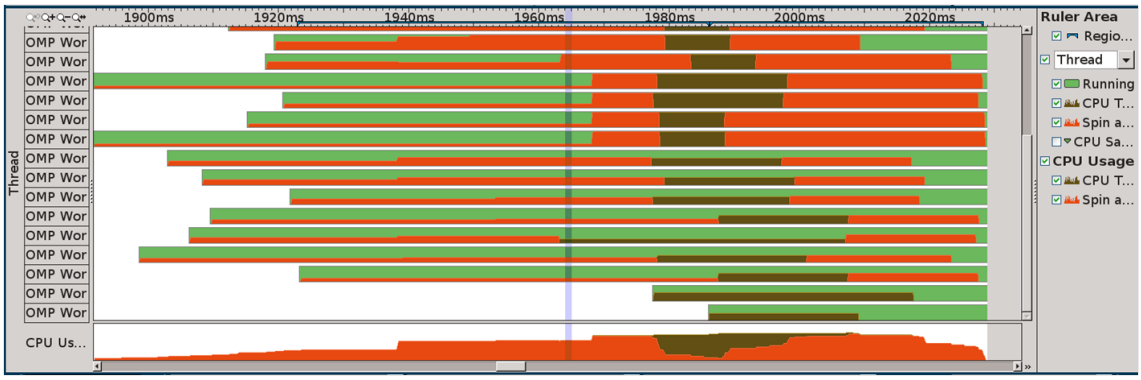
Jak wspomniano w poprzednich rozdziałach, obecne procesory ogólnego przeznaczenia charakteryzują się coraz większą liczbą rdzeni oraz coraz szerszymi rejestrami wektorowymi i jednostkami umożliwiającymi przetwarzanie w modelu SIMD. Wymusza to na programistach uwzględnienie specyfiki architektury oraz korzystanie ze wszelkich dostępnych narzędzi optymalizacyjnych. W ramach niniejszej pracy przetestowano niniejsze procesory z dwoma zadaniami MES - równaniem Poissona oraz uogólnionym równaniem konwekcji-dyfuzji. Oba zadania zostały zaimplementowane z wykorzystaniem zarówno elementów czworościennych jak i pryzmatycznych dla liniowej aproksymacji standardowej oraz z wykorzystaniem elementów pryzmatycznych dla nieciągłej aproksymacji Galerkina. Pozwoliło to na kompleksowe zbadanie możliwości wykorzystania sprzętu.

5.2.2. Liniowa aproksymacja standardowa

W celu uzyskania jak największej wydajności i jak najpełniejszego wykorzystania dostępnych możliwości badanych procesorów przetestowano kilka wersji badanego algorytmu. Pierwsza wersja algorytmu została zaimplementowana z użyciem zrównoleglenia w OpenMP na poziomie pętli po elementach (linijka 4 w Algorytmie 1) i testowana w dwóch wariantach - z optymalizacjami automatycznymi bez wektoryzacji (-O3 -no-vect) oraz z automatyczną wektoryzacją (-O3).

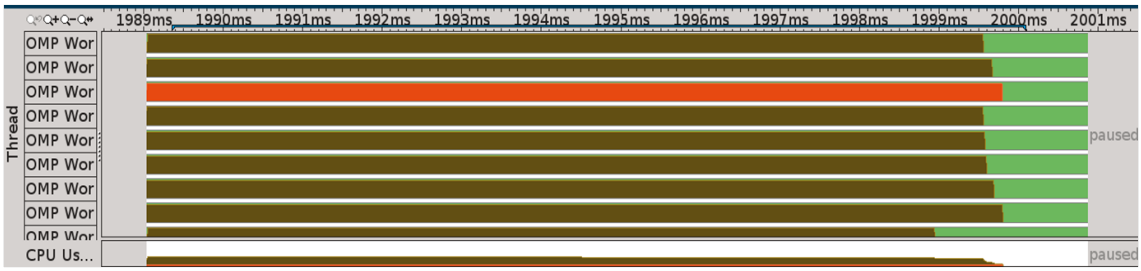
Kolejnym krokiem było dokonanie optymalizacji algorytmu pod kątem architektury. W celu optymalizacji dostępów do pamięci uwzględniono dopasowanie tablic do granicy 32 bajtów oraz, w przypadku elementów pryzmatycznych, rozszerzenie ich wielkości do wielokrotności 4 liczb typu double. W celu ułatwienia wektoryzacji, badana wersja algorytmu została zaimplementowana z użyciem omawianej wcześniej notacji tablicowej Intel Cilk Plus (Rys. 4.2). Równocześnie obszary podlegające wektoryzacji zaznaczano poprzez *#pragma vector* oraz *#pragma simd*. Dodatkowym problemem napotkanym podczas tworzenia tej implementacji, był narzut na tworzenie wątków oraz ich nierównomierne obciążenie (Rys. 5.6). Jak widać na rysunku, wątki uruchamiają się w innym czasie (obszar zielony) i przez większość czasu czekają na podjęcie pracy (obszar pomarańczowy) zanim w końcu przystąpią do właściwych obliczeń (obszar ciemnozielony).

Problem ten został rozwiązany poprzez wywołanie instrukcji tworzenia wątków (*#pragma omp parallel*) na wcześniejszym etapie programu (podczas przygotowywa-



Rysunek 5.6: Analiza uruchomienia wątków

nia danych dla całkowania) oraz ustawienie zmiennych systemowych *KMP_BLOCK--TIME=0* oraz *OMP_WAIT_POLICY=PASSIVE*, dzięki którym, raz stworzone wątki oczekują na przydział zadań. Wynik tych zabiegów obrazuje rysunek 5.7.



Rysunek 5.7: Równomierne obciążenie wątków

Korzystanie ze wszystkich możliwych funkcjonalności procesora wymuszono na kompilatorze używając przełącznika *-xHost*. Przy optymalizacji korzystano z instrukcji zawartych w [46].

Kolejnym zastosowanym stopniem optymalizacji było użycie instrukcji Intel Intrinsics (Rys. 4.3) do bezpośredniej kontroli nad sposobem zwektoryzowania algorytmu. Służy to głownie porównaniu możliwości klasycznych optymalizacji, wraz ze specjalnymi funkcjami kompilatora, do ręcznej kontroli sprzętu.

W związku z tym, że wersja oparta o Intel Intrinsics wydaje się być już niemożliwa do zoptymalizowania w tej postaci algorytmu, w kolejnym kroku, dokonano jego reorganizacji, wymuszając wektoryzację na tym samym poziomie co zrównoleglenie. Dzięki temu, wszystkie operacje powinny być wykonywane w sposób wektorowy powodując teoretyczną czterokrotną redukcję czasu obliczeń (dla rejestrów 256 bitowych i liczb typu double). Wiązało się z to ze zwiększeniem liczby lokalnych tablic na zmienne tymczasowe, oraz skomplikowaną indeksacją tablic wejściowych i wyjściowych, co może wpływać na finalną wydajność. Dodatkowym czynnikiem wpływającym na wydajność może tu być czterokrotne zwiększenie liczby przesyłanych

Algorytm 10: Algorytm całkowania numerycznego dla wektoryzacji na poziomie pętli po elementach

```
- wczytaj tablice ξ^Q i w^Q z danymi całkowania numerycznego;
- wczytaj wartości wszystkich funkcji kształtu oraz ich pochodnych względem
  współrzędnych lokalnych we wszystkich punktach całkowania w elemencie odniesienia;
for iter = 1 to N_EL / STRIDE do
    - wczytaj współczynniki problemowe dla STRIDE elementów (C^iter_STRIDE) ;
    - wczytaj potrzebne dane dotyczące geometrii STRIDE elementów (G^iter_STRIDE) ;
    - (CZWOROŚCIANY) oblicz potrzebne dane transformacji jakobianowych
      dla STRIDE elementów ;
    - zainicjuj macierze sztywności dla STRIDE elementów A^iter_STRIDE i wektory prawej
      strony b^iter_STRIDE ;
    for i_Q = 1 to N_Q do
        - (PRYZMY) oblicz potrzebne dane transformacji jakobianowych dla STRIDE
          elementów ;
        - korzystając z macierzy jakobianowych oblicz pochodne funkcji kształtu
          względem współrzędnych globalnych – φ^iter_STRIDE[i_Q] ;
        - oblicz współczynniki C^iter_STRIDE[i_Q] i D^iter_STRIDE[i_Q] ;
        for i_S = 1 to N_S do
            for j_S = 1 to N_S do
                A^iter_STRIDE[i_S][j_S] += vol ×C^iter_STRIDE[i_Q]×
                                          ×φ^iter_STRIDE[i_Q][i_S] × φ^iter_STRIDE[i_Q][j_S]
            end
        end
        for i_S = 1 to N_S do
            b^iter_STRIDE[i_S] += vol ×D^iter_STRIDE[i_Q] × φ^iter_STRIDE[i_Q][i_S]
        end
    end
    - zapis całej macierzy A^iter_STRIDE oraz wektora b^iter_STRIDE
end
```

danych w czasie jednego pobrania. Równocześnie należy zauważyć, że taki sposób wektoryzacji, aczkolwiek skomplikowany w implementacji, wydaje się być tożsamy ze sposobem wykonania w modelu OpenCL, co zostanie porównane w dalszej części pracy. Zmodyfikowany algorytm nazwany *Stride* prezentuje algorytm 10.

5.2.2.1. Model wykonania i analiza wydajności

W celu opracowania modelu wykonania posłużono się zarówno oficjalnymi charakterystykami badanego sprzętu, jak również wynikami przeprowadzonych testów wydajnościowych. W celu przetestowania badanych architektur użyto zoptymalizowanych pod architektury firmy Intel benchmarków Linpack [22] oraz Stream [85].

Tabela 5.4: Wydajność badanych procesorów [44]

	Wydajność	**Operacje [GFlops]**	**Pamięć [GB/s]**
2×Xeon E5 2620 (Sandy Bridge) 24 Wątki	**Teoretyczna**	240,0	85,2
	Benchmark	200,9	35,2
2×Xeon E5 2699 v.3 (Haswell) 72 Wątki	**Teoretyczna**	1324,8	136,0
	Benchmark	964,8	96,7

Pierwszy z nich służy do obliczenia szybkości obliczeń w liczbie operacji zmiennoprzecinkowych na sekundę (FLOPS). Drugi benchmark mierzy wydajność połączenia pamięć-procesor poprzez wykonywanie serii operacji (copy, scale, add, triad) na dużej ilości danych. Jako reprezentacyjny wybrano tutaj test "triad" ($a(i) = b(i) + q * c(i)$) ze względu na to, że jest on traktowany jako wzorcowy do mierzenia wydajności komputerów wysokiej wydajności. Uzyskane wyniki wraz z wartościami teoretycznymi przedstawia tabela 5.4. Na podstawie uzyskanych w benchmarkach wydajności możemy obliczyć wspomniany wcześniej "machine balance" korzystając ze wzoru (5.1) [83].

$$\frac{\text{Wydajność przetwarzania operacji } [Flops]}{\text{Liczba dostępów do danej na sekundę}} \tag{5.1}$$

Ze względu na używany typ danych (double), uzyskaną w tabeli 5.4 wartość przepustowości pamięci należy podzielić przez jego rozmiar aby otrzymać liczbę dostępów do zmiennej na sekundę. W związku z tym możemy obliczyć, że współczynnik balansu maszyny dla procesora o architekturze Sandy Bridge wynosi **45,66** a dla architektury Haswell wynosi **79,82**. Porównując te wartości z granicznymi wartościami intensywności arytmetycznej zaprezentowanymi w tabeli 3.7 możemy zauważyć że w przypadku architektury Sandy Bridge dla elementów czworościennych czynnikiem ograniczającym jest pamięć. W przypadku elementów pryzmatycznych oraz zadania konwekcji-dyfuzji czynnikiem ograniczającym jest wydajność przetwarzania operacji zmiennoprzecinkowych. W przypadku zadania Poissona, obliczona intensywność jest na granicy w związku z czym, podczas analizy postanowiono brać pod uwagę zarówno wydajność przetwarzania zmiennoprzecinkowego jak i szybkość pamięci. W przypadku architektury Haswell czynnikiem ograniczającym wydajność algorytmu jest wydajność pobierania i zapisu danych z pamięci RAM. Biorąc pod uwagę czynniki ograniczające oraz uzyskane w benchmarkach wydajności, obliczono

Tabela 5.5: Wzorcowe czasy wykonania całkowania numerycznego 1 elementu w *ns* dla procesora Sandy Bridge

	Poisson			**konwekcja-dyfuzja**		
	czworościan	*pryzma*		*czworościan*	*pryzma*	
	M	**M**	**C**	**M**	**M**	**C**
SB	8,18	15,00	15,67	11,82	18,18	23,92
Haswell	2,98	5,46	3,26	4,30	6,62	4,98

teoretyczne czasy wykonania dla algorytmu. W tym celu skorzystano ze wzorów (5.2):

$$\begin{aligned} T_{wyk\_oper} &= \frac{\text{LICZBA\_OPERACJI}}{\text{WYDAJNOŚĆ [FLOPS]}} \\ T_{wyk\_pam} &= \frac{\text{LICZBA\_DOSTĘPÓW x ROZMIAR\_ZMIENNEJ}}{\text{WYDAJNOŚĆ [GB/s]}} \end{aligned} \tag{5.2}$$

Obliczone czasy wykonania dla obu badanych procesorów prezentuje tabela 5.5. W tabeli oznaczono ograniczenie przez wydajność pamięci jako **M** oraz ograniczenie przez wydajność przetwarzania zmiennoprzecinkowego jako **C**.

5.2.2.2. Wyniki

Wyniki z przeprowadzonych testów dla badanych algorytmów i każdego z procesorów przedstawiają tabele 5.6 oraz 5.7. Pogrubione zostały najlepsze z uzyskanych czasów.

W tabelach 5.8 oraz 5.9 zebrano uzyskane przyśpieszenia. W celach testowych na samym początku badany algorytm uruchomiono z opcją -O0 wyłączającą wszystkie optymalizacje. Jak widać samo zastosowanie optymalizacji w postaci zwijania stałych, rozwijania pętli, eliminacji powtarzających się wyrażeń i innych, stosowanych wraz z poziomem automatycznej optymalizacji O3, dało największe przyśpie-

Tabela 5.6: Uzyskane czasy wykonania algorytmu w *ns* dla procesora o architekturze Sandy Bridge

SB	**Poisson**		**konwekcja-dyfuzja**	
	czworościan	*pryzma*	*czworościan*	*pryzma*
O3 no-vec	20,78	82,07	44,78	149,56
O3 vec	18,72	70,90	31,77	140,40
Cilk	24,64	79,19	31,49	112,54
Intrinsic	**18,08**	54,39	31,49	95,19
Stride	18,66	**52,40**	**31,34**	**84,33**

Tabela 5.7: Uzyskane czasy wykonania w *ns* dla procesora o architekturze Haswell

Haswell	**Poisson**		**konwekcja-dyfuzja**	
	czworościan	*pryzma*	*czworościan*	*pryzma*
O3 no-vec	8,56	23,26	12,30	32,90
O3 vec	8,60	18,35	11,79	26,69
Cilk	**8,45**	19,84	11,65	27,92
Intrinsic	8,56	**18,33**	**11,62**	26,97
Stride	8,47	21,57	11,63	**23,58**

Tabela 5.8: Uzyskane przyśpieszenia dla procesora Sandy Bridge

SB	**Poisson**		**konwekcja-dyfuzja**	
	czworościan	*pryzma*	*czworościan*	*pryzma*
Bez Opt	1	1	1	1
O3 no-vec	5,37	6,33	3,14	3,68
O3 vec	1,11	1,16	1,41	1,07
Cilk	0,76	0,90	1,01	1,25
Intrinsic	1,04	1,30	1,01	1,47
Stride	1,00	1,35	1,01	1,66

szenia. Sama wektoryzacja dała przyśpieszenia na poziomie od 7% do 41% dla procesora Sandy Bridge, a dla procesora Haswell ewidentnie zależała od typu używanego elementu - dla pryzm osiągając 27% przyśpieszenia, a dla czworościanów nie uzyskując go praktycznie wcale, co świadczy o totalnym uzależnieniu wykonania od szybkości przetwarzania pamięci. Poszczególne wersje algorytmu - *Cilk*, *Intrinsic* i *Stride* zostały porównane z automatyczną optymalizacją O3 z wektoryzacją. Jak widać w tabeli 5.8, sytuacja z optymalizacji O3 się odwraca i większe przyśpieszenie uzyskano dla elementów pryzmatycznych dla procesora Sandy Bridge. Dla procesora Haswell uzyskano niewielkie przyśpieszenia - w najlepszym przypadku najbardziej wymagającego zadania konwekcji-dyfuzji i elementów pryzmatycznych wynoszące 13%.

Tabela 5.9: Uzyskane przyśpieszenia dla procesora Haswell

Haswell	**Poisson**		**konwekcja-dyfuzja**	
	czworościan	*pryzma*	*czworościan*	*pryzma*
Bez Opt	1	1	1	1
O3 no-vec	3,30	7,05	2,45	3,64
O3 vec	1,00	1,27	1,04	1,23
Cilk	1,02	0,93	1,01	0,96
Intrinsic	1,00	1,00	1,01	0,99
Stride	1,02	0,85	1,01	1,13

Tabela 5.10: Porównanie wydajności procesora o architekturze Haswell z procesorem o architekturze Sandy Bridge

	Poisson		**konwekcja-dyfuzja**	
	czworościan	*pryzma*	*czworościan*	*pryzma*
Bez Opt	3,95	3,17	4,67	4,60
O3 no-vec	2,43	3,53	3,64	4,55
O3 vec	2,18	3,86	2,70	5,26
Cilk	2,92	3,99	2,70	4,03
Intrinsic	2,11	2,97	2,71	3,53
Stride	2,20	2,43	2,70	3,58

Porównując wydajności pomiędzy poszczególnymi procesorami możemy zauważyć, że procesor o architekturze Haswell jest średnio 3 razy szybszy niż procesor o architekturze Sandy Bridge (Tab 5.10). Z zależności tej można wywnioskować, że czynnikiem ograniczającym wydajność jest szybkość pobierania danych z pamięci RAM, ponieważ zgodnie z wynikami uzyskanymi w benchmarkach, pamięć Haswella jest ok. 3 razy szybsza od Sandy Bridge. Dla bardziej skomplikowanych zadań (konwekcja-dyfuzja) i elementów o nieliniowej geometrii (pryzmy) osiąga przyśpieszenie rzędu 5×, co świadczy o wykorzystaniu zarówno jego możliwości obliczeniowych, jak i szybszego pozyskiwania danych z pamięci.

Następnym krokiem w analizie, było porównane uzyskanych wyników z wcześniej obliczonymi wartościami z tabeli 5.5. Wynik tego porównania zaprezentowano w tabelach 5.11 oraz 5.12.

Jak można zauważyć, w większości przypadków osiągnięto zadowalające wyniki sięgające 45% wydajności na architekturze Sandy Bridge oraz 37% na architekturze Haswell.

5.2.2.3. Wnioski

Podsumowując uzyskane wyniki, możemy wywnioskować, iż półautomatyczna wektoryzacja dokonywana przez kompilator przynosi dobre efekty w porównaniu

Tabela 5.11: Uzyskany procent wydajności teoretycznej dla procesora Sandy Bridge

	Poisson		***konwekcja-dyfuzja***	
	czworościan	*pryzma*	*czworościan*	*pryzma*
O3 no-vec	39,37%	19,10%	26,39%	15,99%
O3 vec	43,70%	22,11%	37,19%	17,03%
Cilk	33,20%	19,79%	37,52%	21,25%
Intrinsic	45,23%	28,82%	37,53%	25,12%
Stride	43,84%	29,91%	37,70%	28,36%

Tabela 5.12: Uzyskany procent wydajności teoretycznej dla procesora Haswell

	Poisson		***konwekcja-dyfuzja***	
	czworościan	*pryzma*	*czworościan*	*pryzma*
O3 no-vec	34,80%	23,48%	34,99%	20,12%
O3 vec	34,66%	29,76%	36,51%	24,80%
Cilk	35,25%	27,53%	36,94%	23,71%
Intrinsic	34,81%	29,79%	37,02%	24,55%
Stride	35,18%	25,32%	37,02%	28,08%

z ręczną reorganizacją kodu. Dla zadania całkowania numerycznego, zastosowanie zrównoleglenia po elementach oraz odpowiednich optymalizacji związanych z poziomem O3 przynosi najlepszy efekt. Korzyści ze stosowania wektoryzacji są zależne od zadania oraz architektury, jednakże można się spodziewać, że dla zadań bardziej wymagających obliczeniowo, powinny być większe. Porównywane wersje algorytmu całkowania numerycznego charakteryzuje bardzo dobry wskaźnik przyśpieszenia obliczeń. Należy zauważyć, że w większości przypadków czynnikiem ograniczającym wydajność okazało się pobieranie oraz zapis danych do pamięci. W związku z tym, aby w pełni wykorzystać możliwości badanych procesorów (szczególnie bardzo szybkiego Haswella) postanowiono sprawdzić zadanie o wyższej intensywności arytmetycznej jakim jest całkowanie numeryczne dla wyższych stopni nieciągłej aproksymacji Galerkina.

5.2.3. Nieciągła aproksymacja Galerkina wyższych rzędów

Dla przypadku aproksymacji wyższych rzędów przetestowano uogólnione zadanie konwekcji-dyfuzji dla elementów pryzmatycznych. W założeniach powinno to pozwolić na odpowiednie wysycenie badanych procesorów obliczeniami i w efekcie zaprezentowanie ich wydajności. W ramach badań przetestowano kilka z badanych w poprzednim podrozdziale wersji algorytmu. Jako referencyjny uznano algorytm skompilowany z automatycznymi optymalizacjami (-O3). Ponadto zmodyfikowano algorytmy Cilk oraz Stride dla celów aproksymacji wyższych rzędów.

5.2.3.1. Model wykonania i analiza wydajności

W celu weryfikacji szacowanej liczby operacji dla zadania konwekcji-dyfuzji z nieciągłą aproksymacją Galerkina użyto profilera Intel VTune. W celu uzyskania liczby operacji w profilerze należało zdefiniować własny test polegający na uzyskaniu następujących liczników sprzętowych:

— FP_COMP_OPS_EXE.SSE_PACKED_DOUBLE – liczącego liczbę operacji dla liczb podwójnej precyzji wykonaną na rejestrach wektorowych o szerokości 128 bitów, czyli na dwóch liczbach typu double,
— FP_COMP_OPS_EXE.SSE_PACKED_SINGLE – liczącego liczbę operacji dla liczb pojedynczej precyzji wykonaną na rejestrach wektorowych o szerokości 128 bitów, czyli na czterech liczbach typu float,
— FP_COMP_OPS_EXE.SSE_SCALAR_DOUBLE – liczącego liczbę pojedynczych operacji wykonaną na liczbach typu double w rejestrach wektorowych,
— FP_COMP_OPS_EXE.SSE_SCALAR_SINGLE – liczącego liczbę pojedynczych operacji wykonaną na liczbach typu float w rejestrach wektorowych,
— FP_COMP_OPS_EXE.X87 – liczącego liczbę pojedynczych operacji zmiennoprzecinkowych wykonanych na rejestrach 64 bitowych (liczby typu double),
— SIMD_FP_256.PACKED_DOUBLE – liczącego liczbę operacji dla liczb podwójnej precyzji na rejestrach wektorowych o szerokości 256 bitów, czyli czterech liczbach typu double,
— SIMD_FP_256.PACKED_SINGLE – liczącego liczbę operacji dla liczb pojedynczej precyzji na rejestrach wektorowych o szerokości 256 bitów, czyli ośmiu liczbach typu float.

Należy tu zauważyć iż używane liczniki sprzętowe nie są zbyt dokładne i często dokonują błędnych obliczeń przez co są one niedostępne na architekturze Haswell [84]. W naszym przypadku jednakże uzyskane wyniki są bardzo bliskie szacowanym, co świadczy o dobrym opracowaniu modelu oraz wysokim współczynniku *hit ratio* w opracowanych kodach. W przypadku najniższego stopnia aproksymacji dla algorytmu *Normal* oraz *Cilk*, kompilator był w stanie dokonać nieznacznej redukcji liczby operacji, co nie było możliwe w przypadku wyższych stopni aproksymacji. Algorytm *Stride*, stworzony do celów aproksymacji liniowej, w przypadku aproksymacji wyższych rzędów, charakteryzuje zwiększona liczba operacji. W przypadku aproksymacji wyższych rzędów, liczba rejestrów AVX okazuje się nie wystarczająca i występuje tu tzw. *register spilling* czyli przeładowywanie danych z pamięci L1, co, jak zauważono wcześniej, powoduje nadmiarowe zliczanie operacji.

W przypadku projektowania algorytmu należało również brać pod uwagę wektorowy sposób pobierania danych - w przypadku architektury Sandy Bridge pojedyncze pobranie realizowane jest na 128 bitach danych, a w przypadku Haswella na 256 bitach [57]. Dla najlepiej zoptymalizowanego algorytmu *Cilk* powinno spowodować to prawie dwukrotny spadek liczby pobrań dla starszej architektury oraz czterokrotny dla nowszej.

Na podstawie danych teoretycznych z tabeli 5.4 oraz obliczonych liczby operacji z tabeli 3.4, dokonano obliczenia czasów wykonania korzystając ze wzoru (5.2). Uzyskane wyniki prezentuje tabela 5.13.

Tabela 5.13: Teoretyczne czasy wykonania algorytmu w μs

	Stopień aproksymacji		
	3	*4*	*5*
SB	3,83	21,32	110,26
Haswell	0,80	4,44	22,97

5.2.3.2. Wyniki

Wyniki z przeprowadzonych testów prezentuje tabela 5.14.

Tabela 5.14: Uzyskane czasy wykonania różnych wersji algorytmu na badanych procesorach (w μs)

		Stopień aproksymacji		
		3	*4*	*5*
SB	*O3 vec*	9,88	50,32	307,83
	Cilk	**7,02**	**43,87**	**229,28**
	Stride	17,25	140,99	813,39
Haswell	*O3 vec*	2,08	12,10	68,79
	Cilk	**1,24**	**7,39**	**44,19**
	Stride	3,02	25,71	186,39

Tabela 5.15 obrazuje uzyskane przyśpieszania w stosunku do standardowego algorytmu z automatycznymi optymalizacjami.

Tabela 5.15: Uzyskane przyśpieszenia

		Stopień aproksymacji		
		3	*4*	*5*
SB	*O3 vec*	1,00	1,00	1,00
	Cilk	**1,41**	**1,15**	**1,34**
	Stride	0,57	0,36	0,38
Haswell	*O3 vec*	1,00	1,00	1,00
	Cilk	**1,68**	**1,64**	**1,56**
	Stride	0,69	0,47	0,37

Jak widać z tabeli, algorytm *Cilk* osiąga bardzo dobre przyśpieszenia w stosunku do automatycznej optymalizacji. W przypadku procesora Sandy Bridge przyśpieszenie sięga 41% a w przypadku procesora Haswell aż 68%. Tak jak przypuszczano, wersja *Stride* osiągnęła najniższe wydajności poprzez zbyt dużą liczbę odwołań

Tabela 5.16: Porównanie wydajności procesora o architekturze Haswell z procesorem o architekturze Sandy Bridge

	Stopień aproksymacji		
	3	*4*	*5*
O3 vec	4,75	4,16	4,47
Cilk	5,68	5,94	5,19
Stride	5,71	5,48	4,36

Tabela 5.17: Uzyskany procent wydajności teoretycznej

		Stopień aproksymacji		
		3	*4*	*5*
SB	*O3 vec*	38,78%	42,38%	35,82%
	Cilk	54,57%	48,61%	48,09%
	Stride	22,22%	15,12%	13,56%
Haswell	*O3 vec*	38,37%	36,72%	33,38%
	Cilk	64,53%	60,10%	51,97%
	Stride	26,44%	17,28%	12,32%

do pamięci związaną z register spillingiem. Porównując wydajności pomiędzy procesorami możemy zauważyć, że analogicznie do przypadku aproksymacji liniowej, dla bardziej skomplikowanego zadania, procesor Haswell osiąga 4-6 krotne przyśpieszenie w stosunku do procesora o architekturze Sandy Bridge (Tabela 5.16).

Porównując uzyskane czasy z wydajnością teoretyczną (Tab. 5.17) możemy zauważyć, że w przeciwieństwie do liniowej aproksymacji standardowej, aproksymacja Galerkina jest zadaniem bardziej wymagającym obliczeniowo, co sprzyja pełniejszemu wykorzystaniu możliwości sprzętu (aż do 64% wydajności teoretycznej).

5.2.3.3. Wnioski

Jak można zauważyć z uzyskanych wyników, ten sam algorytm opracowany w ramach liniowej aproksymacji standardowej *Cilk* daje bardzo dobre rezultaty dla zadania o zdecydowanie większej intensywności wykorzystania obliczeń. Dla najlepszego przypadku osiąga on aż ok. 600 GFlops dla procesora o architekturze Haswell. Świadczy to o dobrym stosunku liczby pobrań z pamięci do ilości obliczeń, oraz o optymalnym wykorzystaniu dostępów do pamięci, wykonywanych w postaci 256 bitowych pobrań. Powoduje to także bardzo dobre wykorzystanie dostępnych rejestrów AVX oraz jednostek arytmetyczno-logicznych na nich operujących. Dzięki badaniu nieliniowej aproksymacji Galerkina na nowoczesnych procesorach ogólnego przeznaczenia potwierdzono wniosek, iż algorytm całkowania numerycznego dla bardziej skomplikowanych zadań oraz wyższych stopni aproksymacji, jest zadaniem wystarczająco

wymagającym aby wykorzystać możliwości współczesnego sprzętu przeznaczonego do obliczeń.

5.2.4. Wnioski końcowe

Dla procesorów CPU kluczowym czynnikiem ograniczającym wydajność w przypadku algorytmu całkowania numerycznego jest szybkość uzyskiwania oraz zapisu danych z i do pamięci RAM. Dzięki dokonanym optymalizacjom udało się uzyskać bardzo dobre wyniki wydajnościowe badanego algorytmu. Dla zadania o większej intensywności arytmetycznej, udało się dobrze wykorzystać możliwości obliczeniowe badanych procesorów, osiągając ponad 60% wydajności teoretycznej. Należy zauważyć, że w wielu przypadkach kluczowym elementem wymagającym zoptymalizowania w algorytmach numerycznych jest sposób dostępu do pamięci. Pomóc tu może wykorzystanie nowoczesnych funkcji języków programowania takich jak Cilk w C/C++ lub reorganizacja algorytmu wykorzystująca charakterystyki danego sprzętu np. *Stride*. W obu przypadkach należy brać pod uwagę ilość danych potrzebnych do pobrania, gdyż wymuszenie maksymalizacji użycia rejestrów może prowadzić do register spillingu znacząco obniżającego finalną wydajność.

5.3. Akceleratory GPU

5.3.1. Metodologia

W ramach badań dokonanych na akceleratorach opartych o budowę procesorów graficznych, dokonano implementacji algorytmu całkowania numerycznego z liniową aproksymacją standardową dla elementów czworościennych i pryzmatycznych oraz dwóch zadań – prostego zadania Poissona oraz uogólnionego zadania konwekcji-dyfuzji.

W projektowaniu algorytmu dla akceleratorów opartych o budowę kart graficznych, ważną kwestią jest wzięcie pod uwagę wielopoziomowej struktury pamięci tego typu urządzeń. Jako że, struktura języków OpenCL i CUDA, służących do programowania tego typu akceleratorów, oparta jest na budowie GPU, to projektowanie algorytmu należy zacząć od bezpośredniego przydziału poszczególnych jego części odpowiednim poziomom pamięci. Dodatkowym czynnikiem wpływającym na organizację pracy jest model wykonania w językach OpenCL/CUDA, który dzieli pracę pomiędzy wątki, pogrupowane w grupy robocze dzielące ten sam obszar pamięci wspólnej. Wymusza to na programiście różne sposoby organizacji pracy w zależności od wymaganych zasobów. W przypadku algorytmu całkowania numerycznego można zastosować kilka strategii podziału pracy:

— Jeden Element – Jeden Wątek (tożsamy z podziałem na CPU) – jeden element jest przetwarzany przez jeden wątek naraz – jeden wątek może przetwarzać kilka kolejnych elementów.
— Jeden Element – Kilka Wątków – jeden element przetwarzany jest przez kilka wątków – dodatkowy poziom zrównoleglenia na poziomie pierwszej pętli po funkcjach kształtu (linijka 12 w Algorytmie 1)
— Jeden Element – Jedna Grupa Robocza – jeden element przetwarzany przez cała grupę roboczą – może występować podział kolumnami macierzy sztywności (linijka 12 w Algorytmie 1) lub podział szachownicowy względem obu pętli i_S oraz j_S.
— Jeden Element – Kilka Kerneli – dane potrzebne do obliczenia macierzy sztywności jednego elementu są obliczane w kilku osobnych kernelach – np. dane transformacji Jakobianowych (linijka 9 w Algorytmie 1) mogą być liczone osobno dla wszystkich elementów, a następnie wykorzystywane przez kernel liczący macierze sztywności.

W każdym przypadku konieczne jest przygotowanie danych po stronie kodu CPU, a następnie przesłanie ich do GPU. Danymi tymi są:

1. Parametry sterujące procedury całkowania numerycznego - ilość elementów liczona przez wątek (grupę roboczą) oraz całkowita liczba elementów,
2. Dane punktów całkowania - współrzędne oraz wagi dla elementu referencyjnego,
3. Dane funkcji kształtu (i ich pochodnych) dla elementu referencyjnego,
4. Dane geometryczne - współrzędne wierzchołków obliczanego elementu,
5. Współczynniki problemowe – różne dla każdego punktu Gaussa w przypadku równania Poissona lub stałe dla elementu w przypadku uogólnionego zadania konwekcji-dyfuzji.

Dane 4 i 5 te grupowane są element po elemencie, a następnie przesyłane do pamięci akceleratora. Argumentem wyjściowym procedury jest gotowa tablica zawierająca elementowe macierze sztywności oraz wektory prawej strony.

5.3.2. Liniowa aproksymacja standardowa

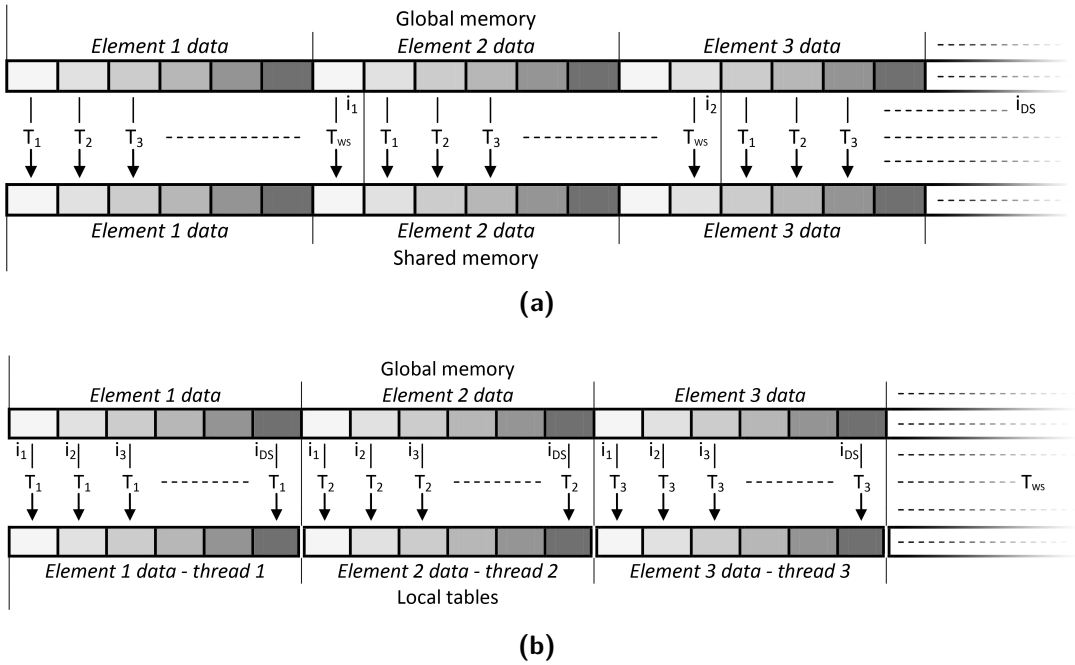
W przypadku algorytmu całkowania numerycznego dla liniowej aproksymacji standardowej zastosowano pierwszy z prezentowanych w poprzednim podrozdziale podziałów pracy - Jeden Element – Jeden Wątek. Wynika to z tego, iż w przypadku tego typu aproksymacji ilość potrzebnych danych oraz obliczeń jest stosunkowo mała (Tablice 3.3 oraz 3.6). Nowy zmodyfikowany algorytm całkowania numerycznego (QSS) dla akceleratorów opisuje Algorytm 11. Wersje SQS, SSQ i wariant dla elementów czworościennych różnią się analogicznie do wersji zaprezentowanych w Algorytmach 4, 6 oraz 3. Ze względu na różne możliwości korzystania z pamięci w akceleratorach opartych o budowę kart graficznych wyodrębniono 9 parametrów algorytmu całkowania numerycznego, mających wpływ na jego finalną wydajność:

1. WORKGROUP_SIZE - rozmiar grupy roboczej - związany z minimalną ilością wątków możliwych do uruchomienia na akceleratorze - w przypadku testów na GPU, został ustawiony na 64 ze względu na zalecenia producentów badanych kart.
2. USE_WORKSPACE_FOR_PDE_COEFF - współczynnik decydujący czy dane problemowe mają być przechowywane w pamięci wspólnej wątków czy w rejestrach (linijka 4 Algorytmu 11).
3. USE_WORKSPACE_FOR_GEO_DATA - dane geometryczne przechowywane w pamięci wspólnej wątków lub w rejestrach (linijka 3 Algorytmu 11).

Algorytm 11: Uogólniony algorytm całkowania numerycznego na akceleratory

```
1  - określenie parametrów algorytmu – N_EL, N_Q, N_S;
2  for e = 1 to N_EL do
3      - wczytaj potrzebne dane dotyczące geometrii elementu (Tablica G^e);
4      - wczytaj współczynniki problemowe wspólne dla wszystkich punktów całkowania (Tablica C^e);
5      - zainicjuj elementową macierz sztywności A^e i elementowy wektor prawej strony b^e;
6      for i_Q = 1 to N_Q do
7          - wczytaj współrzędne punktu całkowania oraz jego wagę z tablicy z danymi całkowania
             numerycznego;
8          - oblicz potrzebne dane transformacji jakobianowych (∂x/∂ξ, ∂ξ/∂x, vol);
9          if COMPUTE_ALL_SHAPE_FUN_DER then
10             for i_S = 1 to N_S do
11                 - korzystając z macierzy jakobianowych oblicz pochodne funkcji kształtu względem
                     współrzędnych globalnych;
12             end
13         end
14         for i_S = 1 to N_S do
15             - oblicz współczynniki problemowe C[i_Q] i D[i_Q] w punkcie całkowania;
16             if (COMPUTE_ALL_SHAPE_FUN_DER) then
17                 - wczytaj uprzednio obliczone pochodne funkcji kształtu;
18             else
19                 - korzystając z macierzy jakobianowych oblicz pochodne funkcji kształtu względem
                     współrzędnych globalnych;
20             end
21             for i_D = 0 to N_D do
22                 b^e[i_S] += vol ×D[i_Q][i_D] × φ[i_Q][i_S][i_D];
23             end
24             for j_S = 1 to N_S do
25                 if (COMPUTE_ALL_SHAPE_FUN_DER) then
26                     - wczytaj uprzednio obliczone pochodne funkcji kształtu;
27                 else
28                     - korzystając z macierzy jakobianowych oblicz pochodne funkcji kształtu
                         względem współrzędnych globalnych;
29                 end
30                 for i_D = 0 to N_D do
31                     for j_D = 0 to N_D do
32                         A^e[i_S][j_S] += vol ×C[i_Q[[i_D][j_D]×
33                                                ×φ[i_Q][i_S][i_D] × φ[i_Q][j_S][j_D];
34                     end
35                 end
36             end
37         end
38     end
39     - zapis całej macierzy A^e oraz wektora b^e do bufora wyjściowego w pamięci globalnej akceleratora;
40 end
```

4. USE_WORKSPACE_FOR_SHAPE_FUN - dane funkcji kształtu i ich pochodnych przechowywane w pamięci wspólnej wątków lub w rejestrach (linijki 10-12, 18-20 oraz 27-29 Algorytmu 11).

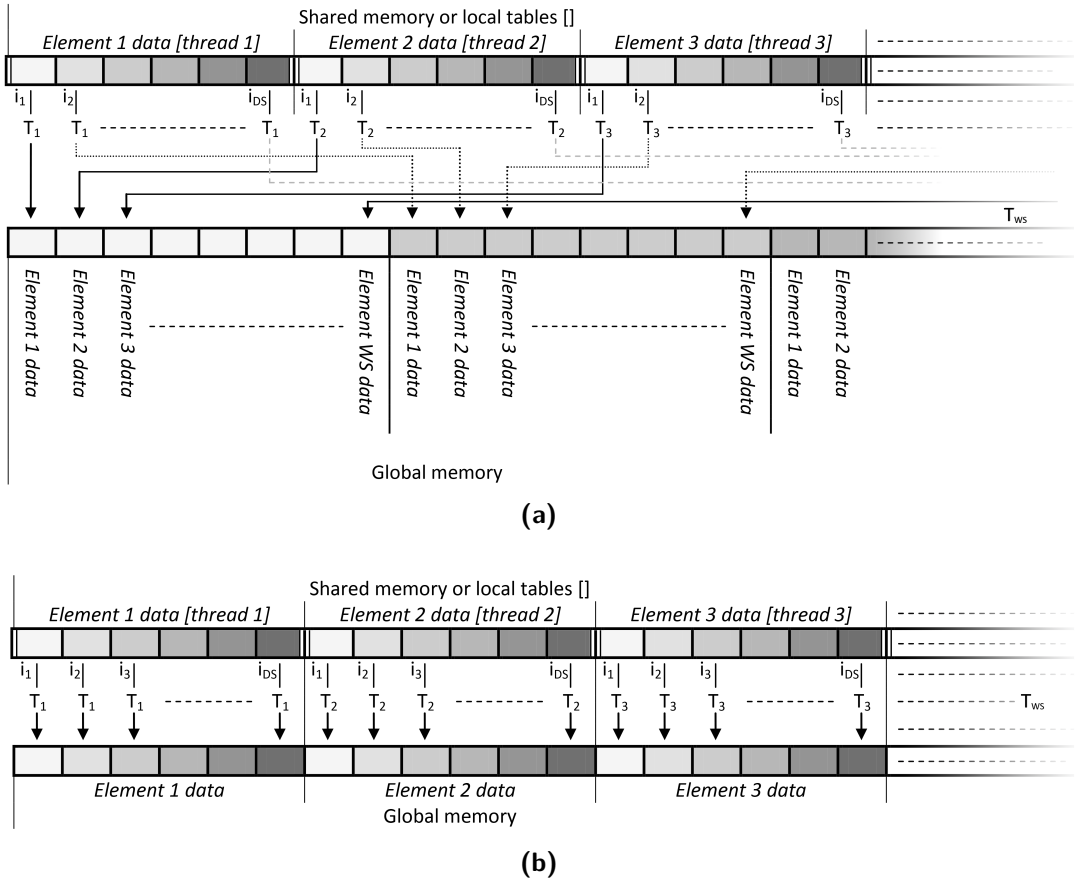


Rysunek 5.8: Organizacja odczytu danych z pamięci: a) odczyt ciągły (coalesced), b) odczyt nieciągły (non-coalesced). Kolejne wątki w grupie roboczej o rozmiarze *WS*, zostały oznaczone jako $\mathrm{T_j}, \mathrm{j} = 1, .., \mathrm{WS}$, a kolejne iteracje czytania danych o rozmiarze *DS* przez pojedyncze wątki, zostały oznaczone jako $\mathrm{i_k}, \mathrm{k} = 1, .., \mathrm{DS}$.

5. USE_WORKSPACE_FOR_STIFF_MAT - lokalne macierze sztywności i wektory prawej strony przechowywane w pamięci wspólnej wątków lub w rejestrach (linijki 5, 22, 32-33 oraz 39 Algorytmu 11).
6. PADDING - dodatkowe przesunięcie w celu wyeliminowania błędów typu "bank-conflict" (podczas początkowej definicji tablic).
7. COMPUTE_ALL_SHAPE_FUN_DER - obliczanie wartości wszystkich funkcji kształtu i ich pochodnych przed wejściem do podwójnej pętli po funkcjach kształtu (linijki 9,16 oraz 25 Algorytmu 11).
8. COAL_READ - czytanie w sposób ciągły (Rys. 5.8).
9. COAL_WRITE - zapis w sposób ciągły (Rys. 5.9).

Dwa ostatnie parametry algorytmu całkowania numerycznego, dotyczą sposobu zapisu oraz odczytu danych z pamięci akceleratora. W związku z fizyczną budową tego typu urządzeń i działania wątków w grupach, sposób dostępu do pamięci może mieć znaczący wpływ na wydajność algorytmu. Optymalny dostęp do pamięci powinien być pogrupowany względem wątków [94], dzięki czemu zapisu i odczytu dokonujemy w sposób ciągły (coalesced) - całymi wektorami danych z pamięci.

Tablice przekazywane jako argumenty procedury całkowania numerycznego (np. dane geometryczne oraz współczynniki problemowe), mogą być używane bezpośrednio



Rysunek 5.9: Organizacja zapisu danych do pamięci: a) zapis ciągły (coalesced), b) zapis nieciągły (non-coalesced). Kolejne wątki w grupie roboczej o rozmiarze *WS*, zostały oznaczone jako $\mathrm{T_j}, \mathrm{j} = 1, .., \mathrm{WS}$, a kolejne iteracje zapisu danych o rozmiarze *DS* przez pojedyncze wątki, zostały oznaczone jako $\mathrm{i_k}, \mathrm{k} = 1, .., \mathrm{DS}$.

w obliczeniach, lub wcześniej pobrane do lokalnych tablic, przechowywanych w rejestrach lub pamięci wspólnej wątków. Ze względu na minimalizację liczby dostępów do pamięci globalnej urządzenia, oraz jak najlepszą optymalizację dostępu do pamięci, została wybrana ta druga strategia. W celu zapewnienia ciągłego dostępu do pamięci w ramach jednej grupy procesów, dane różnych elementów powinny być umieszczone w kolejnych komórkach pamięci DRAM, czyniąc dane pojedynczego elementu rozrzuconymi po całej tablicy wejściowej. Jednakże takie podejście może skutkować zbyt dużą ingerencją w istniejący kod, jak również spowodować spadek wydajności w momencie przygotowywania tychże danych wejściowych. W związku z tym zastosowano odwrotną strategię - dane wejściowe ułożone są w sposób naturalny - tj. dane jednego elementu są umieszczone razem, jednakże zmienia się sposób ich odczytu. W tym przypadku musimy użyć pamięci współdzielonej jako tymczasowego bufora odczytu danych, ze współdzielonym dostępem dla wszystkich wątków w ramach grupy roboczej. Dzięki temu, dane czytane są w sposób ciągły - pojedynczy

wątek czyta kolejną komórkę pamięci i zapisuje do współdzielonego bufora jak przedstawiono na rysunku 5.8. Tak odczytane dane mogą być zostawione w buforze i użyte w późniejszym czasie do obliczeń, lub mogą być przepisane do rejestrów, zwalniając bufor do dalszego użycia. Sposób przechowywania tychże danych w odpowiednich buforach definiują parametry USE_WORKSPACE_FOR_*.

Dla danych wyjściowych, możliwe stało się zastosowanie innej strategii i bezpośredni zapis w postaci otrzymywanej bezpośrednio z wątków. Dzięki temu, nie używamy dodatkowego bufora pamięci wspólnej i dokonujemy bezpośredniego zapisu danych w sposób ciągły, jak przedstawiono na rysunku 5.9. Z tego powodu, podczas dalszego korzystania z danych w procedurach agregacji oraz solwera, musimy uwzględnić zmienioną organizację pamięci - co w przypadku przekazania obliczeń innemu kernelowi na akceleratorze tego samego typu, może być traktowane jako istotna zaleta.

Lista parametrów algorytmu 11 dotyczy w większości sposobu obsługi pamięci w modelu OpenCL, który, jak wspominano wcześniej, bazuje na architekturze kart graficznych. Łączna liczba kombinacji tych parametrów dla dowolnej architektury wynosi 80. W związku z tym, że dla każdej z architektur istnieją 3 warianty algorytmu opisane w rozdziale 3.5 (QSS, SQS i SSQ), oraz mamy 2 zadania i 2 typy elementów, to łączna liczba kombinacji na architekturę wynosi 960. Z tego względu, koniecznym stało się opracowanie systemu automatycznego tuningu parametrycznego algorytmu całkowania numerycznego na dowolną architekturę. System ten składa się z zestawu skryptów oraz specjalnych fragmentów kodu odpowiadających za kompilację kernela z odpowiednimi opcjami. W związku z różnymi opcjami kompilacji dostępnymi u różnych dostawców oprogramowania, koniecznym stało się zaimplementowanie różnych opcji zależnych od zainstalowanego akceleratora oraz wersji frameworka OpenCL. Dodatkowo, system automatycznego tuningu, ma też zaimplementowaną opcję generacji, oraz wstępnej analizy kodu assemblera dla architektur najbardziej popularnych dostawców (Intel, Nvidia, AMD).

Dla celów głębszej analizy wykonania, oraz w związku z koniecznością użycia narzędzia Nvidia Visual Profiler dla akceleratora Tesla K20m, dokonano przepisania istniejącej implementacji OpenCL na język CUDA. Aby sprawdzić czy kod w OpenCL jest równoważny wersji CUDA dla analizowanych przypadków, dokonano porównania wygenerowanych plików w języku PTX (Parallel Thread eXecution Instruction Set), nie stwierdzając istotnych różnic. Pozwoliło to na głębszą analizę kodu uruchamianego na rozwiązaniach firmy Nvidia, z równoczesnym zachowaniem przenośności kodu w OpenCL.

5.3.2.1. Model wydajności

W nowszych architekturach, oprócz standardowego podziału na pamięć współdzieloną (shared/local), stałą (constant) oraz globalną (global), należy brać również pod uwagę obecność pamięci cache (najczęściej dwupoziomowej). Czyni to dużą cześć wykonania niedeterministyczną i zbliża się w ogólnej konstrukcji do standardowego modelu znanego z CPU. Z tego względu nie jest możliwym dokonanie szczegółowej analizy dostępów do poszczególnych poziomów pamięci GPU. Jednakże przy projektowaniu algorytmu, należy brać pod uwagę obecność tych szybkich poziomów pamięci oraz fakt, że dostęp do nich jest bezpośrednio zależny od programisty.

Tabela 5.18: Charakterystyki badanych akceleratorów [91, 112]

	Tesla K20m	**Radeon R9280X**
Rdzenie (multiprocesory)	13	32
Cache L2	1536 kB	768 kB
Pamięć globalna (DRAM)	5GB	3GB
Charakterystyka rdzenia		
Jednostki SIMD (SP/DP)	192/64	64/16
Cache L1	16 kB	16 kB
Rejestry	256 kB	256 kB
Pamięć stała	48 kB	64 kB
Pamieć współdzielona	48 kB	32 kB

Tabela 5.18 obrazuje podstawowe parametry badanych akceleratorów. Jak widać z tabeli, stosunkowo duża liczba rejestrów dla architektur opartych na budowie GPU, okupiona jest małym rozmiarem szybkich poziomów pamięci (szczególnie cache L2). Nieodpowiedni przydział zasobów może tu skutkować zmniejszeniem ilości równocześnie uruchomionych wątków, lub w najgorszym przypadku uniemożliwić uruchomienie. Parametry wydajnościowe badanych akceleratorów, teoretyczne oraz uzyskane we wcześniej opisywanych testach Stream oraz Linpack, wykorzystywane w opracowaniu modelu wydajności prezentuje tabela 5.19. Tak jak w przypadku procesorów CPU, możliwym jest obliczenie maksymalnego balansu maszyny dla liczb podwójnej precyzji, który dla Radeona wynosi **23,74** a dla Tesli **61,11**. Porównując uzyskane wartości z granicznymi wartościami intensywności arytmetycznej poszczególnych wersji algorytmu z tabeli 3.7 możemy zauważyć że w przypadku karty Tesla i elementów czworościennych, czynnikiem ograniczającym wydajność jest szybkość przetwarzania danych z pamięci. W przypadku elementów pryzmatycznych dla algorytmów SQS oraz SSQ, liczba obliczeń już dominuje nad pobraniami, sprawiając że czynnikiem ograniczającym staje się wydajność przetwarzania zmiennoprzecinkowego. W przypadku karty Radeon jedynie zadanie Poissona dla elementów

czworościennych jest ograniczone wydajnością pamięci. Pozostałe zadania ogranicza szybkość przetwarzania zmiennoprzecinkowego, z tym zastrzeżeniem że dla zadania konwekcji-dyfuzji i elementów czworościennych w algorytmach QSS i SQS, wynik intensywności arytmetycznej znajduje się na granicy balansu procesora, co czyni koniecznym sprawdzenie obu opcji dla tego typu zadania.

Mnogość opcji rozważanej implementacji czyni niemożliwym opracowanie szczegółowego modelu dla każdego z możliwych 80 wariantów algorytmu. W przypadku rozważania algorytmu na architekturze opartej o budowę kart graficznych, jest to wręcz niekonieczne. Dla tego typu konstrukcji musimy założyć zapis i odczyt w sposób ciągły, a każdą z opcji przechowywania danych w pamięci shared, możemy rozważać osobno.

Tabela 5.19: Wydajność badanych akceleratorów [91, 112]

	Radeon	Tesla
Teoretyczna wydajność obliczeń podwójnej precyzji [TFlops]	1.13	1.17
Wydajność uzyskana w benchmarku DGEMM	0.65	1.10
Teoretyczna wydajność obliczeń pojedynczej precyzji [TFlops]	4.51	3.52
Wydajność uzyskana w benchmarku SGEMM	1.70	2.61
Teoretyczna przepustowość pamięci [GB/s]	288	208
Przepustowość pamięci z benchmarku STREAM	219	144

W celu opracowania odpowiednich wersji procedury całkowania, koniecznym jest porównanie możliwości badanych akceleratorów z zapotrzebowaniem na pamięć dla każdego z wariantów algorytmu, rozważanym w rozdziale 3.5 oraz przedstawionym w tabeli 3.6. Użycie pamięci współdzielonej może okazać się koniecznym w związku z niewystarczającą liczbą rejestrów dla kernela. W przypadku przekroczenia liczby używanych rejestrów, zmienne lokalne dla wątków, są przechowywane w cache oraz pamięci DRAM (register spilling). W związku z tym, że pamięć shared jest kilkukrotnie szybsza niż pamięć DRAM, odpowiednie jej użycie ma bardzo duży wpływ na wydajność. Wobec tego, że na akceleratorach GPU, minimalna ilość wątków jest zwyczajowo duża i w naszym przypadku wynosi 64, należy mieć na uwadze, że przechowanie n zmiennych lokalnych dla każdego wątku wymagać będzie $n \times 64$ miejsc w pamięci wspólnej. Z tego względu, w rozważaniach dotyczących opracowania modelu wydajności, autor bierze pod uwagę przechowywanie jedynie po jednej z używanych tablic w pamięci wspólnej (współczynniki problemowe, dane geometryczne, dane funkcji kształtu lub finalne macierze $\boldsymbol{A}^e$ i $\boldsymbol{b}^e$).

Na podstawie danych z tabeli 3.7 oraz danych z tabeli 5.19, można obliczyć szacowany czas potrzebny na wykonanie poszczególnych algorytmów zaprezentowany w tabeli 5.20. Analogicznie do przypadku CPU w tabeli zaprezentowano czasy wynikające z ograniczenia wydajności algorytmu szybkością przetwarzania danych

Tabela 5.20: Wzorcowe czasy wykonania (w *ns*) całkowania numerycznego pojedynczego elementu w zależności od czynnika ograniczającego

		Poisson			**konwekcja-dyfuzja**			
		czworościan	*pryzma*		*czworościan*		*pryzma*	
		M	**M**	**C**	**M**	**C**	**M**	**C**
R280X	**QSS**	**1,32**	2,41	**4,85**	1,90	**1,97**	2,92	**7,39**
	SQS	**1,32**	2,41	**16,62**	1,90	**1,97**	2,92	**19,16**
	SSQ	**1,32**	2,41	**88,06**	1,90	**4,70**	2,92	**98,36**
K20m	**QSS**	**2,00**	**3,67**	2,86	**2,89**	1,16	**4,44**	4,37
	SQS	**2,00**	3,67	**9,82**	**2,89**	1,16	4,44	**11,32**
	SSQ	**2,00**	3,67	**52,04**	**2,89**	2,78	4,44	**58,12**

z pamięci (**M**) oraz szybkością przetwarzania zmiennoprzecinkowego (**C**). Dodatkowo pogrubiono finalne najwyższe czasy, będące uzyskanymi czasami granicznymi.

5.3.2.2. Transfer z i do pamieci hosta

Zagadnieniem, którym należałoby się zająć w przypadku badań akceleratorów z osobną pamięcią w stosunku do gospodarza (np. podłączonych przez interfejs PCI-express), jest transfer danych między pamięciami globalnymi tych urządzeń. W przypadku całkowania numerycznego, traktowanego jako osobna procedura, oczywistym wydaje się fakt, że przesył danych silnie zdominuje całkowity czas wykonania. Z tego też względu, badania poświęcone czasowi przesyłu danych zostaną podjęte w dalszej części niniejszej pracy, w trakcie testowania przyszłościowego rozwiązania w postaci jednostki GPU zintegrowanej z CPU. Badania autora mają na celu wykazanie, że nawet tak mało wymagający obliczeniowo algorytm jakim jest całkowanie numeryczne, będzie potrafił skorzystać z zalet nowoczesnych architektur obliczeniowych. Równocześnie autor ma nadzieję, że w przyszłych tego typu konstrukcjach, problem przesyłu danych pomiędzy akceleratorem a pamięcią gospodarza zostanie całkowicie wyeliminowany na wzór obecnych rozwiązań z architektur hybrydowych.

5.3.2.3. Wyniki

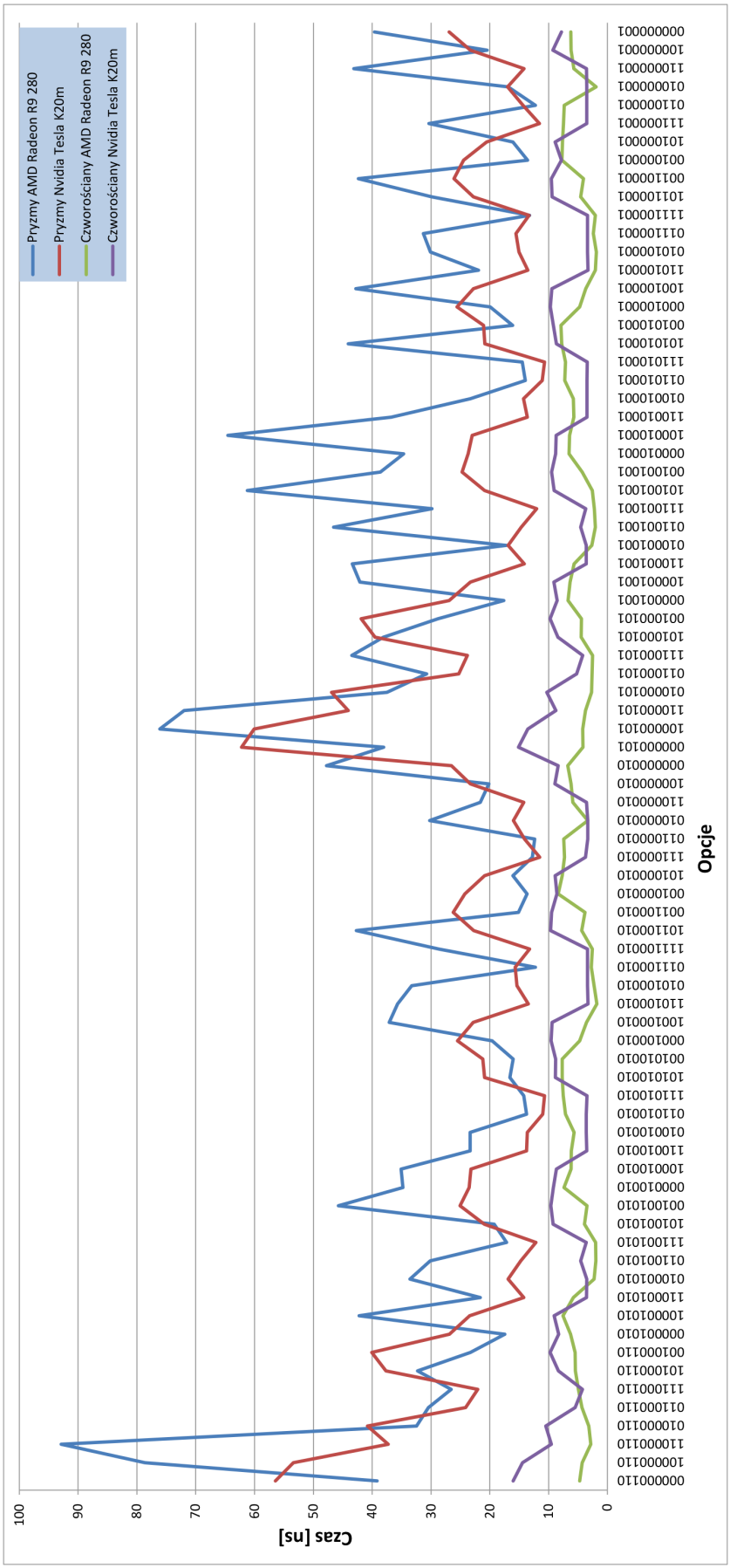
Wynik działania automatycznego tuningu algorytmu QSS na przykładzie problemu Poissona przedstawia rysunek 5.10. Opcje tuningu zostały ułożone w następującej kolejności:

1. COAL_READ (CR)
2. COAL_WRITE (CW)
3. COMPUTE_ALL_SHAPE_FUN_DER (CASFD)
4. USE_WORKSPACE_FOR_PDE_COEFF (PDE)

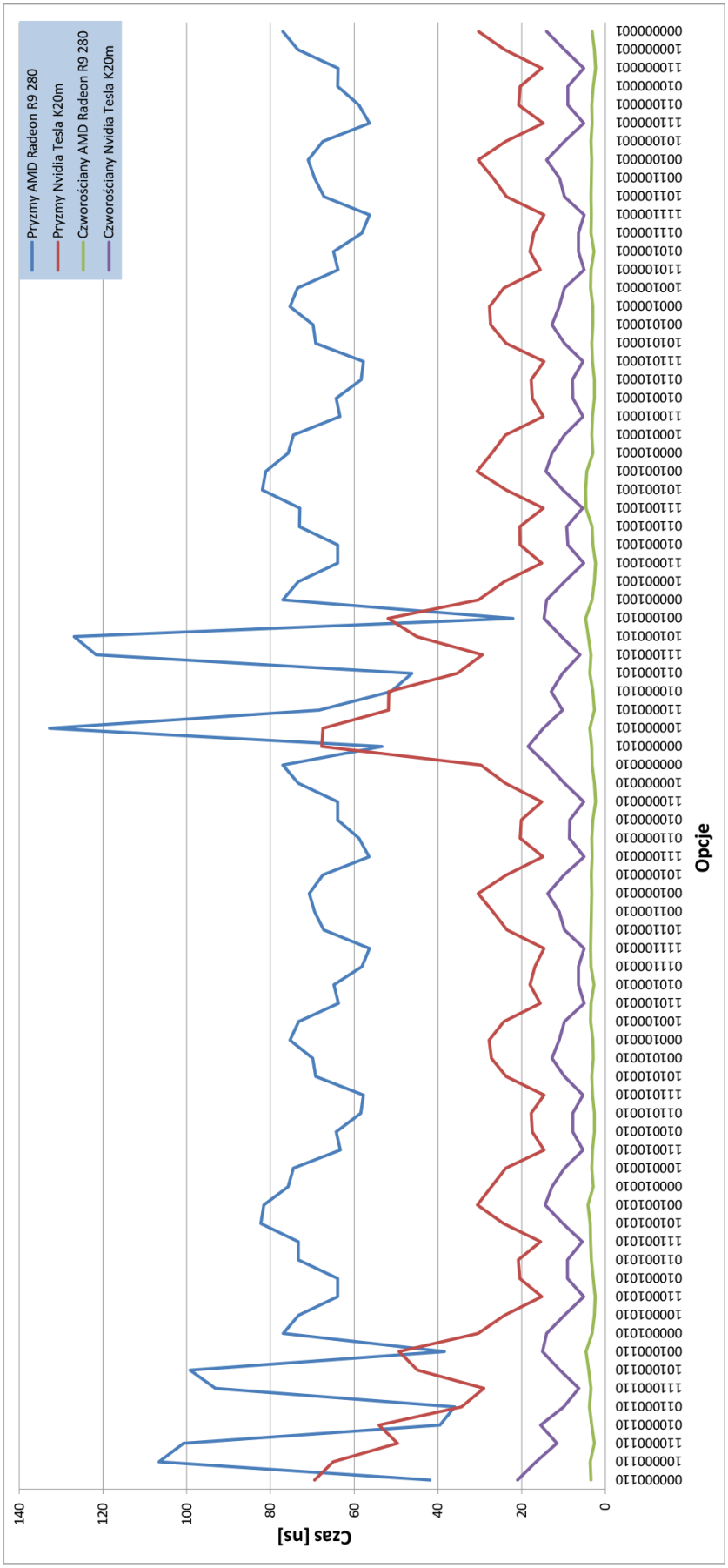
5. USE_WORKSPACE_FOR_GEO_DATA (GEO)
6. USE_WORKSPACE_FOR_SHAPE_FUN (SHP)
7. USE_WORKSPACE_FOR_STIFF_MAT (STIFF)
8. PADDING=0
9. PADDING=1

Na osi poziomej wykresu 5.10 zaznaczona jest kombinacja opcji w kolejności podanej wcześniej - w przypadku włączenia danej opcji zaznaczane jest przez 1 a w przypadku wyłączenia 0. Pierwszą zauważalną rzeczą na wykresie, jest jego chaotyczność związana z tym, iż kombinacja jednych opcji dla danej architektury może skutkować zupełnie innym wynikiem na drugiej architekturze. Kolejną zauważalną rzeczą jest powtarzalność wykresu względem opcji paddingu – szczególnie dla architektury firmy Nvidia. Świadczy to o tym, że w trakcie wykonania programu raczej nie występują błędy "bank conflict". Dla karty Radeon najlepszą opcją dla elementów pryzmatycznych okazało się nie używanie pamięci współdzielonej, włączenie obliczania pochodnych funkcji kształtu przed pętlami po funkcjach kształtu oraz wyłączenie odczytu ciągłego, przy zachowaniu ciągłego zapisu. Dla karty Tesla oraz tego samego typu elementów, opcja ciągłego odczytu wydaje się kluczowa i dlatego w tym przypadku najlepszy czas uzyskała kombinacja z ciągłym odczytem, zapisem oraz włączoną opcją CASFD. Dodatkowo, kwestia użycia pamięci wspólnej dla niewielkiej ilości danych geometrycznych, wydaje się nie mieć wpływu na ogólny wynik. Najgorsze wyniki uzyskano dla przechowywania danych wyjściowych algorytmu (STIFF) w pamięci współdzielonej. Dla elementów czworościennych sytuacja jest trochę inna, gdyż uzyskano najlepsze wyniki dla układu parametrów z włączonym ciągłym odczytem i zapisem, wyłączoną opcją CASFD oraz użyciem pamięci wspólnej dla współczynników problemowych. Także w tym przypadku, wyłączenie opcji ciągłego odczytu ma większy wpływ dla architektury firmy Nvidia. Najgorsze wyniki dla czworościanów, uzyskano dla nieciągłego odczytu oraz pamięci wspólnej użytej do przechowywania macierzy wynikowej (STIFF) dla karty Tesla, a także przy przeliczaniu pochodnych funkcji kształtu przed głównymi pętlami dla karty R9 280X. Co ciekawe, o wiele (ok. 10 razy) droższa karta Nvidii jest średnio 30% lepsza od karty AMD dla elementów geometrycznie wieloliniowych, a dla elementów liniowych jest średnio 40% gorsza, co potwierdza szacowania teoretyczne.

Dla zadania konwekcji-dyfuzji wyniki działania algorytmu QSS prezentuje rysunek 5.11. Tak jak w przypadku zadania Poissona, zadanie konwekcji-dyfuzji charakteryzuje powtarzalność wyników względem opcji dopasowania tablic. Dla karty Radeon oraz elementów pryzmatycznych, uzyskano najlepsze wyniki nie korzystając z zapisu i odczytu ciągłego, przy równoczesnym używaniu pamięci współdzielonej



Rysunek 5.10: Automatyczny tuning parametryczny dla zadania Poissona oraz algorytmu QSS



Rysunek 5.11: Automatyczny tuning parametryczny dla zadania konwekcji-dyfuzji oraz algorytmu QSS

dla finalnej macierzy sztywności. Na Tesli uzyskano takie same wyniki jak dla zadania Poissona - najlepsze dla zapisu i odczytu ciągłego, obliczania pochodnych funkcji kształtu przed podwójną pętlą po funkcjach kształtu oraz użycia pamięci wspólnej dla danych geometrycznych. W przypadku obu kart najgorsze wyniki uzyskano wyłączając zapis ciągły i używając pamięci wspólnej do przechowywania finalnych macierzy (STIFF) przy równoczesnym wyłączeniu opcji CASFD. Dla elementów czworościennych, najlepsze wyniki uzyskano przy nie używaniu pamięci wspólnej i wyłączeniu opcji CASFD dla kart Radeon. W przypadku karty Tesla, włączenie opcji CASFD okazało się konieczne, razem z użyciem pamięci wspólnej do przechowywania współczynników problemowych. Najgorszy wynik na Tesli uzyskano w przypadku wyłączenia ciągłego zapisu i odczytu oraz użyciem pamięci wspólnej dla przechowywania finalnych macierzy (STIFF). Dla Radeona, najgorsze wyniki uzyskano wyłączając ciągły zapis oraz używając pamięci wspólnej do przechowywania pochodnych funkcji kształtu obliczanych przed podwójną pętlą po N_S.

Szczegółowe wyniki uzyskane z systemu automatycznego tuningu dla algorytmów SQS oraz SSQ i badanych zadań znajdują się w dodatku A na końcu niniejszej pracy.

Tabela 5.21: Zestawienie najlepszych wyników (w ns) dla poszczególnych wersji algorytmu całkowania numerycznego

		Poisson		**konwekcja-dyfuzja**	
		czworościan	*pryzma*	*czworościan*	*pryzma*
Radeon	**QSS**	1,824	**12,238**	**2,371**	**22,105**
	SQS	**1,791**	48,421	2,826	56,285
	SSQ	1,805	64,290	2,910	57,251
Tesla	**QSS**	3,291	**10,687**	5,055	**14,633**
	SQS	2,909	33,534	**4,434**	54,667
	SSQ	**2,881**	72,476	4,447	149,559

Zestawienie uzyskanych najlepszych wyników z systemu automatycznego tuningu parametrycznego, prezentuje tabela 5.21. Jak widać z tabeli, dla elementów czworościennych zmiana kolejności pętli po punktach całkowania oraz funkcjach kształtu, przyniosła niewielką poprawę wyników dla akceleratora Nvidia Tesla (ok. 13% zysku). Dla Radeona R9 280X zysk jest praktycznie niezauważalny. Wersja SSQ okazała się gorsza od wersji SQS w przypadku Radeona o ok. 3% bez względu na geometrię elementu, a dla Tesli aż 63% wolniejsza dla pryzm. Dla elementów czworościennych wyniki dla akceleratora firmy Nvidia są porównywalne.

Podsumowując uzyskane z systemu automatycznego tuningu parametrycznego wyniki, można zauważyć, że odpowiedni dobór parametrów ma bardzo duże znaczenie. Dla rozwiązania firmy Nvidia koniecznym wydaje się użycie opcji ciągłego

Tabela 5.22: Zestawienie najlepszej kombinacji opcji dla poszczególnych wersji algorytmu całkowania numerycznego

		Poisson		**konwekcja-dyfuzja**	
		czworościan	*pryzma*	*czworościan*	*pryzma*
Radeon	**QSS**	1101000	**0110000**	**1100000**	**0010001**
	SQS	**1110100**	0010000	0110100	0110000
	SSQ	1100100	0100010	0100100	0100100
Tesla	**QSS**	1101000	**1110100**	1111000	**1110100**
	SQS	0110100	0100001	**1111000**	1110010
	SSQ	**1110001**	0101000	1110001	1100000

zapisu. Dla architektury AMD, w większości przypadków ciągły zapis także pomagał, jednakże dla niektórych opcji nie mógł być on łączony z ciągłym odczytem. Dla algorytmu SQS obowiązkowym wydaje się włączenie opcji CASFD, bez względu na typ elementu. Tymczasem dla algorytmu SSQ, opcja ta powinna być włączona wyłącznie dla czworościanów na Tesli. Użycie pamięci wspólnej zmienia się w zależności od dostępnych zasobów – dla elementów czworościennych w algorytmach SQS i SSQ dobre wyniki przynosiło użycie tej pamięci dla przechowywania danych geometrycznych. Zestawienie kombinacji najlepszych opcji (bez opcji Padding – pominiętej jako nieistotna) prezentuje tabela 5.22.

Porównanie uzyskanych wyników z tabeli 5.21 z oszacowaniami teoretycznymi z tabeli 5.20, umieszczono w tabeli 5.23. W tabeli pogrubiono wyniki z najlepszymi czasami wykonania dla każdego typu zadania. Jak można zauważyć dla algorytmu SSQ osiągnięto lepsze niż zakładane teoretycznie czasy wykonania dla Radeona R9 280X. Z tego faktu można wywnioskować, że w przypadku tego procesora kompilator optymalizujący, dokonał znaczącej redukcji liczby redundantnych obliczeń charakterystycznych dla algorytmu SSQ. Jest to widoczne szczególnie w przy-

Tabela 5.23: Uzyskany procent wydajności teoretycznej dla badanych akceleratorów GPU

		Poisson		**konwekcja-dyfuzja**	
		czworościan	*pryzma*	*czworościan*	*pryzma*
Radeon	**QSS**	72,09%	**39,60%**	**83,07%**	**33,45%**
	SQS	**73,43%**	34,31%	69,69%	34,05%
	SSQ	72,87%	136,97%	161,55%	171,81%
Tesla	**QSS**	60,77%	**34,31%**	57,15%	**30,37%**
	SQS	68,74%	29,28%	**65,15%**	20,71%
	SSQ	**69,41%**	71,80%	64,97%	38,86%

padku zadania konwekcji-dyfuzji gdzie uzyskane czasy wykonania dla algorytmów SQS i SSQ są bardzo bliskie.

Porównując uzyskane wyniki między poszczególnymi akceleratorami widzimy, iż dla zadania wymagającego obliczeniowo (czyli konwekcji-dyfuzji) różnica wydajności wynosi 34%. Jednakże dla zadań prostszych, jak zadanie Poissona, różnica już jest niewielka i wynosi 13%. Dla zadania Poissona z elementami pryzmatycznymi, akcelerator AMD jest szybszy od akceleratora Nvidii o 38% a dla zadania konwekcji-dyfuzji aż o 47%.

Analiza wykonania dokonana przy pomocy profilera Nvidii wykazuje że w przypadku elementów czworościennych i zadania Poissona wykorzystanie GPU (occupancy) jest limitowane przez użycie pamięci wspólnej i jest na poziomie 24.8%. W przypadku wykorzystania elementów pryzmatycznych, profiler wskazuje że czynnikiem limitującym wydajność są opóźnienia operacji oraz pamięci spowodowane przez niską zajętość karty (12,5%). Jednakże wszystkie próby sztucznego zwiększenia czynnika *occupancy* poprzez zmniejszenie użycia rejestrów lub pamięci wspólnej, powodowały wzrost czasu wykonania i obniżenie ogólnej wydajności. Podobną sytuację mamy dla elementów czworościennych i bardziej wymagającego obliczeniowo zadania konwekcji-dyfuzji. W tym przypadku czynnikiem limitującym zajętość akceleratora jest użycie pamięci wspólnej. Powoduje to stosunkowo niski współczynnik *occupancy* równy 12,5%. Jednakże jak widać z wykresu 5.11, zmiany ilości wykorzystania pamięci wspólnej, przy zachowaniu innych parametrów algorytmu, nie powodowały wzrostu wydajności. Taką samą sytuację obserwujemy dla zadania konwekcji-dyfuzji dla elementów pryzmatycznych. Czynnikiem limitującym wydajność jest tutaj użycie rejestrów wpływające na zajętość akceleratora. Także i w tym przypadku można stwierdzić, że żadna próba ograniczenia użycia zasobów nie spowodowała podniesienia ogólnej wydajności algorytmu.

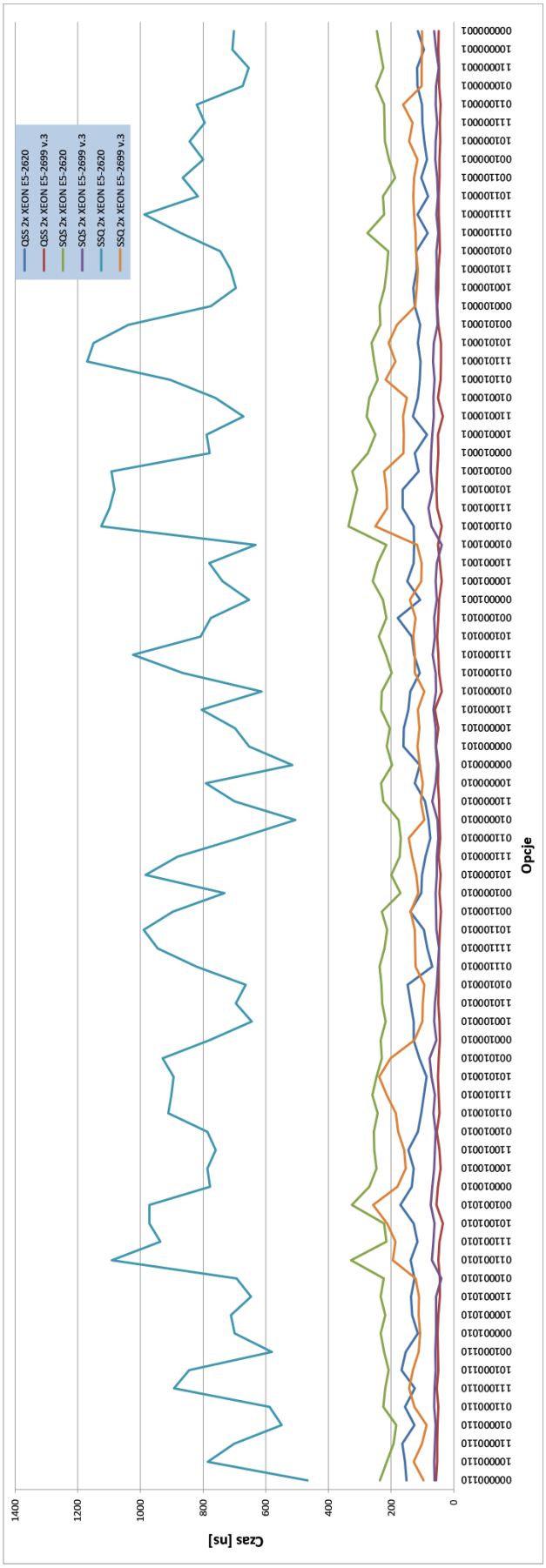
W przypadku akceleratora AMD Radeon R9 280X, dla najlepszego wyniku dla zadania Poissona i elementów czworościennych uzyskano zajętość na takim samym poziomie, jak w przypadku akceleratora Nvidia, czyli 25%. Jest ono limitowane przez użycie pamięci wspólnej. W przypadku elementów pryzmatycznych, znacząco rośnie liczba użytych rejestrów (z 63 do 231). Powoduje to zajętość karty na poziomie 10%. Tak jak w przypadku karty Tesla, sztuczne ograniczanie ilości użytych rejestrów, nie powodowało wzrostu wydajności. Mimo teoretycznego nie wykorzystywania pamięci wspólnej (wyłączone coal read), zajętość tej pamięci wynosi 144 bajty (18 liczb typu double). W przypadku algorytmu SSQ możemy również zauważyć zajętość na poziomie 10%, limitowaną przez użycie 237 rejestrów na wątek. Takie samo jest wykorzystanie pamięci wspólnej, co świadczy o mniejszym wykorzystaniu

pamięci wspólnej niż zakładana w oszacowaniach teoretycznych. Świadczy to o wysokim stosunku wykorzystania rejestrów, co znacząco poprawia wydajność. Dla zadania konwekcji–dyfuzji i elementów czworościennych, przeciwnie do akceleratora Tesla, czynnikiem limitującym zajętość jest użycie rejestrów. W porównaniu do poprzednich przypadków ich użycie jest stosunkowo niskie (141 na wątek), jednakże ogranicza liczbę równocześnie uruchomionych grup roboczych do 4 z 40 możliwych. W tym przypadku, obserwujemy również wysokie wykorzystanie pamięci wspólnej, które zbliża się do czynnika limitującego zajętość karty, ograniczając liczbę uruchomionych grup wątków do 6. W przypadku najbardziej wymagającego zadania konwekcji–dyfuzji dla elementów pryzmatycznych, mimo uzyskania bardzo dobrych wyników, współczynnik zajętości akceleratora wynosi 7,5%. W tym przypadku czynnikiem ograniczającym jest użycie pamięci wspólnej. Jest to zrozumiałe, gdyż w przypadku najlepszego wyniku dla karty Radeon, wygrała opcja z użyciem pamięci wspólnej na przechowywanie macierzy sztywności. Dla przypadku SSQ możemy po raz drugi zauważyć maksymalizację wykorzystania rejestrów (248 na wątek). Ogranicza to współczynnik occupancy do 10%, ale daje wyraźny wzrost wydajności.

Aby ocenić otrzymane wyniki wydajnościowe, dokonano ich porównania z uzyskanymi wcześniej wynikami dla procesorów ogólnego przeznaczenia (CPU).

5.3.2.4. Porównanie wyników

W celu porównania wyników otrzymanych na akceleratorach GPU z badanymi wcześniej procesorami CPU, uruchomiono system automatycznego tuningu dla testowanych procesorów Xeon E5 2620 oraz Xeon E5 2699 v.3. W przypadku ustawień OpenCL skorzystano z instrukcji w [45] i ustalono rozmiar grupy roboczej na 8. Jako reprezentatywny wybrano przykład zadania Poissona oraz elementów pryzmatycznych (Rys. 5.12). Tak jak w poprzednim wypadku, szczegółowe wykresy z wynikami dla pozostałych wersji algorytmu można znaleźć w dodatku B na końcu niniejszej pracy. Na wykresie możemy zaobserwować większą chaotyczność wyników niż w przypadku GPU. Wynika to z tego, że dla procesorów CPU model pamięci OpenCL jest przybliżany poprzez podział pamięci globalnej na sektory z pamięcią stałą i wspólną. W związku z tym, poszczególne tablice przechowywane są w tak samo szybkiej pamięci, a jedyną różnicę możemy zaobserwować dzięki użyciu pamięci cache. Bardzo jednorodne wyniki dla architektury Haswell, świadczą o tym, że algorytm całkowania numerycznego idealnie wpasował się w bardzo dużą pamięć cache L3 badanego procesora, dzięki czemu pomiędzy poszczególnymi opcjami w OpenCL nie ma dużej różnicy. W przypadku procesora Sandy Bridge niewielka ilość cache nie wystarczała dla wszystkich przypadków - szczególnie widoczne jest to dla elementów pryzmatycznych, gdzie algorytm SSQ znacząco odstaje od reszty wyników



Rysunek 5.12: Automatyczny tuning parametryczny dla zadania Poissona i elementów pryzmatycznych na procesorach CPU

Tabela 5.24: Zestawienie najlepszych wyników (w *ns*) dla poszczególnych wariantów algorytmu całkowania numerycznego z tuningiem parametrycznym

		Poisson		**konwekcja-dyfuzja**	
		czworościan	*pryzma*	*czworościan*	*pryzma*
SB	**QSS**	21,576	**69,077**	29,494	**86,744**
	SQS	**20,653**	168,444	**27,926**	163,705
	SSQ	21,129	466,350	29,198	585,764
Haswell	**QSS**	8,312	**34,069**	12,004	**34,988**
	SQS	**7,879**	38,736	**11,330**	45,680
	SSQ	8,076	86,336	11,765	97,978

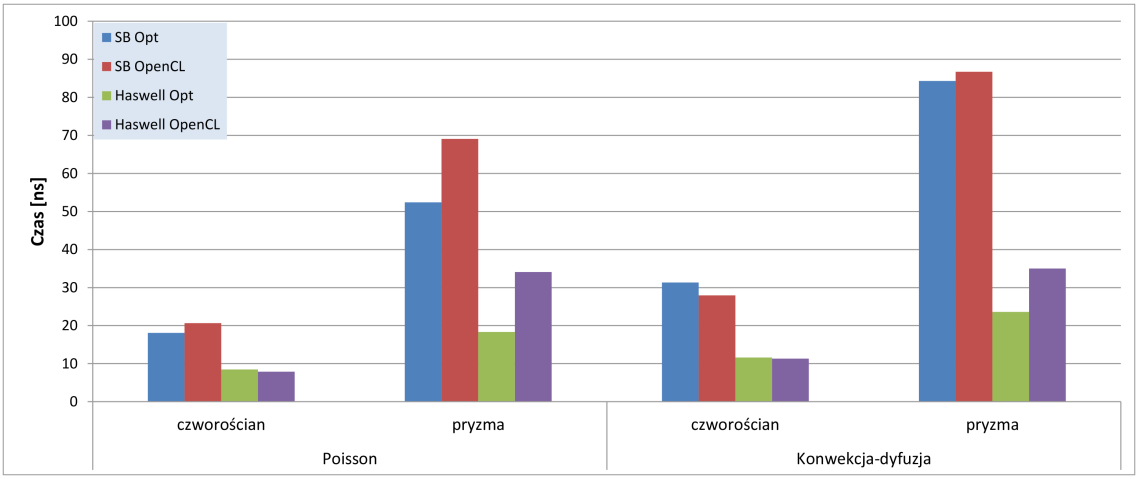
Tabela 5.25: Zestawienie najlepszej kombinacji opcji dla poszczególnych wersji algorytmu całkowania numerycznego na CPU

		Poisson		**konwekcja-dyfuzja**	
		czworościan	*pryzma*	*czworościan*	*pryzma*
SB	**QSS**	010000010	**011100010**	001100010	**011100001**
	SQS	**011100010**	011000010	**100010010**	111000001
	SSQ	000000001	000000110	111010001	000000010
Haswell	**QSS**	111000001	**101001010**	001100001	**001010001**
	SQS	**010010010**	010001001	**010000010**	110001010
	SSQ	011000110	010000110	111000010	010000010

- zapewne poprzez swoje zapotrzebowanie na pamięć. Dla elementów czworościennych, możemy zauważyć grupowanie się wyników względem architektury - widzimy tu prawie trzykrotny wzrost wydajności dla architektury Haswell.

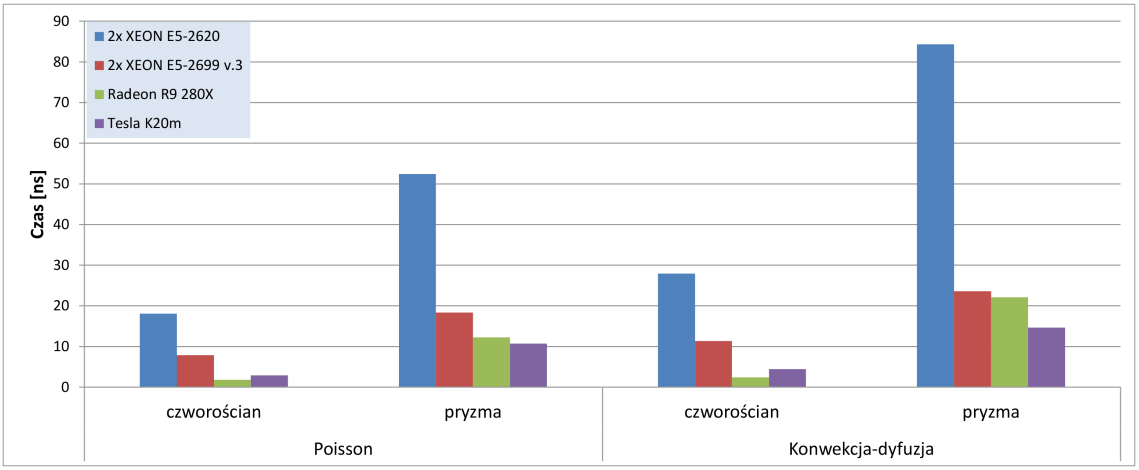
Zestawienie najlepszych wyników z tuningu parametrycznego na procesorach CPU obrazuje tabela 5.24. Jak widać z tabeli, najlepsze wyniki dla elementów czworościennych uzyskano dla wariantu SQS algorytmu, zarówno dla architektury Sandy Bridge jak i Haswell. Z tabeli 5.25 widzimy, że kombinacje opcji wpływających na wydajność są dosyć chaotyczne. Dzięki temu, możemy zauważyć dużą przydatność systemu automatycznego tuningu, który jest w stanie zoptymalizować uruchomienie kodu OpenCL nawet na architekturze, która swoją budową i działaniem odbiega od modelu działania akceleratorów GPU.

Porównując uzyskane wyniki z autotuningu oraz z ręcznych optymalizacji kodu zaprezentowanych w podrozdziale 5.2 możemy zauważyć bardzo zbliżone wyniki (Rys. 5.13). W przypadku elementów czworościennych w trzech przypadkach uzyskano lepsze czasy w OpenCL, niż w przypadku korzystania z ręcznych optymalizacji. W przypadku elementów pryzmatycznych oraz procesora Haswell udało się uzyskać prawie dwa razy lepsze rezultaty korzystając z ręcznych optymalizacji. Jednakże należy zauważyć, że system automatycznego tuningu potrafi bardzo dobrze



Rysunek 5.13: Porównanie wyników uzyskanych dla algorytmu całkowania numerycznego na procesorach ogólnego przeznaczenia

dopasować uruchamiany kod do możliwości architektury, co pozwala na uzyskanie satysfakcjonujących wyników nawet dla architektur odbiegających od optymalnych.



Rysunek 5.14: Porównanie wyników uzyskanych dla algorytmu całkowania numerycznego dla akceleratorów GPU oraz procesorów CPU

Na wykresie 5.14 porównano wyniki uzyskane dla obu badanych typów urządzeń - CPU oraz GPU. Możemy tu zauważyć, iż starsza architektura CPU Sandy Bridge znacząco odstaje od pozostałych. Jednakże w przypadku architektury Haswell, wyniki nie są już tak rażące. W przypadku elementów geometrycznie liniowych, procesor Haswell okazuje się być prawie 4,5 raza wolniejszy od karty Radeon R9 280X oraz 2,5 raza wolniejszy od karty Tesla K20m. W przypadku elementów pryzmatycznych, Haswell jest niewiele wolniejszy od Radeona R9 280X (7% – 30%) oraz około 40% wolniejszy niż Tesla K20m.

5.3.2.5. Wnioski

Uzyskane wyniki dla akceleratorów opartych na budowie kart graficznych świadczą o ich ogromnym potencjale obliczeniowym. Dzięki systemowi automatycznego tuningu, uzyskano lepsze wyniki niż dla zoptymalizowanych wersji algorytmu całkowania numerycznego na nowoczesnych procesorach CPU. Uzyskany wysoki procent wykorzystania możliwości obliczeniowych badanych akceleratorów pozwala stwierdzić że zadanie całkowania numerycznego z liniową aproksymacją standardową okazuje się wystarczająca by wysycić możliwości badanych architektur. Uzyskane wyniki procentowego wykorzystania architektur pozwalają stwierdzić że w przypadku zadań o mniejszej intensywności arytmetycznej z elementami geometrycznie liniowymi, algorytm całkowania numerycznego jest w stanie osiągnąć imponujące 83% wydajności teoretycznej dla karty Radeon oraz 69% dla karty Tesla. Dzięki przeprowadzonym badaniom można zauważyć, że coraz to nowsze procesory, wyposażone w dużą ilość rdzeni, jednostki wektorowe i operatory FMA, coraz bardziej zbliżają się wydajnością do akceleratorów GPU. Wraz z problemem przesyłu danych na akceleratorów GPU poprzez wolny interfejs PCI-Express, który zostanie poruszony później, może to stawiać pod znakiem zapytania opłacalność tego typu rozwiązań. W przypadku akceleratorów bazujących na procesorach graficznych kluczowym czynnikiem wpływającym na finalną wydajność okazuje się być minimalizacja dostępów do pamięci współdzielonej wraz z maksymalizacją użycia rejestrów. Należy to mieć na uwadze przy wyborze tego typu architektur w celu przenoszenia konkretnych algorytmów, gdyż użycie pamięci shared jest równocześnie jedynym sposobem na synchronizację wątków, co w niektórych typach algorytmów może okazać się problematyczne.

5.4. Intel Xeon Phi

5.4.1. Metodologia

W przypadku testowania koprocesora Intel Xeon Phi o architekturze Knights Corner (KNC), postanowiono skorzystać z doświadczeń uzyskanych przy programowaniu procesorów CPU oraz akceleratorów GPU. Koprocesor Intel Xeon Phi jest reklamowany jako narzędzie pozwalające w łatwy sposób przyśpieszyć istniejące kody korzystające z OpenMP i półautomatycznej wektoryzacji. Kluczem do uzyskania wysokiej wydajności na badanym koprocesorze ma być jej uzyskiwanie na procesorach Intel Xeon, jednakże jak wykazały badania innych autorów ([115], [114], [96], [28]) wymaga to dodatkowych optymalizacji. Na badanym koprocesorze postanowiono uruchomić zoptymalizowane wersje algorytmu uzyskane dla procesorów Intel Xeon – *Cilk* oraz *Stride*. W obu przypadkach zmieniono dopasowanie (*aligment*) danych z 32 na 64 bajty aby w pełni wykorzystać dostępne 512 bitowe jednostki wektorowe koprocesora. Dodatkowo w wersji *Stride*, przetestowano rozmiar pasma równy 8. Dodatkowym czynnikiem wpływającym na wydajność była konieczność ustawienia zmiennych systemowych KMP_AFFINITY="granularity=fine,compact" oraz MIC_USE_2MB_-BUFFERS=2M. W celu uruchomienia zadań na koprocesorze skorzystano z modelu *offload*, wysyłając wcześniej przygotowane dane przed uruchomieniem algorytmu i pobierając je po jego zakończeniu. Pozwoliło to w stosunkowo łatwy sposób, rozszerzyć istniejący kod o potrzebne instrukcje, bez zmiany jego struktury.

W przypadku wersji OpenCL, ustalono rozmiar grupy roboczej na wielokrotność 16 zgodnie z zaleceniami z [48]. Jest to związane z tym że model programowania OpenCL dla tego koprocesora definiuje rozmiar grupy SIMD na 16 bez względu na używany typ danych. Pozwala to na automatyczną wektoryzację i ma znacząco redukować czas przetwarzania. Tak jak w przypadku CPU, poziomy pamięci z OpenCL

Tabela 5.26: Charakterystyki badanych koprocesorów [54]

	5110P	**7120P**
Rdzenie	60	61
Wątki	236	240
Cache L2	30 MB	30,5 MB
Pamięć globalna (DRAM)	8GB	16GB
Charakterystyka rdzenia		
Cache L1	32 kB	32 kB
Rejestry	2 kB	2 kB

są mapowane na pamięć koprocesora. Charakterystyki sprzętowe badanych procesorów pokazuje tabela 5.26. Jak można zauważyć, w porównaniu z architekturami GPU, ilość rejestrów jest stosunkowo mała. Znacząco za to wzrosła ilość dostępnej pamięci cache L1 oraz L2. Może to wpłynąć na sposób implementacji wersji OpenCL, jednakże dzięki systemowi automatycznego tuningu, powinno być możliwe dobranie optymalnych parametrów.

5.4.2. Liniowa aproksymacja standardowa

5.4.2.1. Model wydajności

W celu opracowania modelu wydajności posłużono się charakterystykami wydajnościowymi badanych koprocesorów zaprezentowanymi w tabeli 5.27. Porównując wydajność procesorów Xeon Phi z badanymi wcześniej procesorami Xeon (Tab. 5.4), możemy zauważyć, że chociaż dla starszej architektury, szacowane teoretyczne wydajności są o wiele wyższe, tak dla nowszej architektury Haswell, badane koprocesory nie uzyskują już przewagi. Świadczy to o tym, że architektura KNC wydaje się być już zbyt przestarzała w porównaniu z nowymi architekturami CPU. Obliczone współczynniki równowagi maszyny wynoszą **37,28** dla wersji 5110P oraz **44,15** dla 7120P.

Tabela 5.27: Wydajność badanych koprocesorów [53, 28]

	Wydajność	**Operacje [GFlops]**	**Pamięć [GB/s]**
5110P	**Teoretyczna**	1011	320
	Benchmark	769	165
7120P	**Teoretyczna**	1208	352
	Benchmark	999	181

Porównując uzyskane wartości z wartościami intensywności arytmetycznej dla algorytmu obliczonymi w tabeli 3.7 możemy zauważyć że w przypadku architektury KNC oba procesory są, w przypadku zadań z elementami pryzmatycznymi, ograniczone przez szybkość obliczeń. W przypadku elementów czworościennych, czynnikiem ograniczającym wydajność jest pamięć RAM za wyjątkiem algorytmu SSQ dla zadania konwekcji-dyfuzji którego intensywność arytmetyczna jest wyższa niż balans obu maszyn.

Na podstawie danych z tabeli 5.27 (dane z benchmarków) oraz obliczonych teoretycznych liczb operacji oraz dostępów, obliczono spodziewany czas wykonania algorytmu, który zaprezentowano w tabeli 5.28.

Tabela 5.28: Oczekiwany czas wykonania (w *ns*) algorytmu całkowania numerycznego dla poszczególnych jego wersji

		Poisson			**konwekcja-dyfuzja**			
		czworościan	*pryzma*		*czworościan*		*pryzma*	
		M	**M**	**C**	**M**	**C**	**M**	**C**
5120P	**QSS**	**1,75**	3,20	**4,10**	**2,52**	1,66	3,88	**6,25**
	SQS	**1,75**	3,20	**14,04**	**2,52**	1,66	3,88	**16,20**
	SSQ	**1,75**	3,20	**74,43**	2,52	**3,97**	3,88	**83,14**
7120P	**QSS**	**1,59**	2,92	**3,15**	**2,30**	1,28	3,54	**4,81**
	SQS	**1,59**	2,92	**10,81**	**2,30**	1,28	3,54	**12,47**
	SSQ	**1,59**	2,92	**57,30**	2,30	**3,06**	3,54	**64,00**

Tak jak w przypadku wcześniejszych badań, literami **M** oraz **C** oznaczono czasy ograniczone przez szybkość pobrań i zapisów z/do pamięci oraz szybkość przetwarzania operacji zmiennoprzecinkowych.

5.4.2.2. Wyniki

Uzyskane wyniki dla wersji z wykorzystaniem OpenMP oraz modelu offload prezentuje tabela 5.29. Jak widać z tabeli, najlepsze czasy wykonania uzyskano dla koprocesora 5110P oraz wersji z pasmem o szerokości 8. Co ciekawe, teoretycznie szybszy koprocesor 7120P uzyskał gorsze rezultaty w 67% przypadków. W jego przypadku zaobserwowano też większą chaotyczność wyników, które wbrew oczekiwaniom nie wskazują na algorytm *Stride=8* jako najlepszy.

Oczekiwany procent obliczonej wydajności teoretycznej prezentuje tabela 5.30. Jak można zauważyć, uzyskane wyniki są bardzo niskie w stosunku do oczekiwanych i sięgają ledwie 5,4% teoretycznej wydajności.

W celach porównawczych uruchomiono również opracowany wcześniej tuning parametryczny algorytmu w OpenCL. Jako reprezentatywne wybrano zadanie Po-

Tabela 5.29: Uzyskane wyniki (w *ns*) dla badanych wersji algorytmu całkowania numerycznego z wykorzystaniem OpenMP i modelu offload

		Poisson		**konwekcja-dyfuzja**	
		czworościan	*pryzma*	*czworościan*	*pryzma*
5110P	*Cilk*	58,612	212,531	68,498	220,966
	Stride=4	56,470	166,289	61,502	170,292
	Stride=8	**51,796**	**145,313**	**46,597**	**138,664**
7120P	*Cilk*	**45,996**	312,725	67,682	**170,609**
	Stride=4	57,240	**209,099**	**59,070**	175,347
	Stride=8	60,748	292,382	66,664	243,964

Tabela 5.30: Uzyskany procent wydajności teoretycznej dla badanych koprocesorów Xeon Phi dla implementacji z użyciem modelu offload

		Poisson		**konwekcja-dyfuzja**	
		czworościan	*pryzma*	*czworościan*	*pryzma*
5110P	*Cilk*	2,98%	1,93%	3,68%	2,83%
	Stride=4	3,09%	2,46%	4,10%	3,67%
	Stride=8	3,37%	2,82%	5,41%	4,51%
7120P	*Cilk*	3,46%	1,01%	3,40%	2,82%
	Stride=4	2,78%	1,51%	3,89%	2,74%
	Stride=8	2,62%	1,08%	3,45%	1,97%

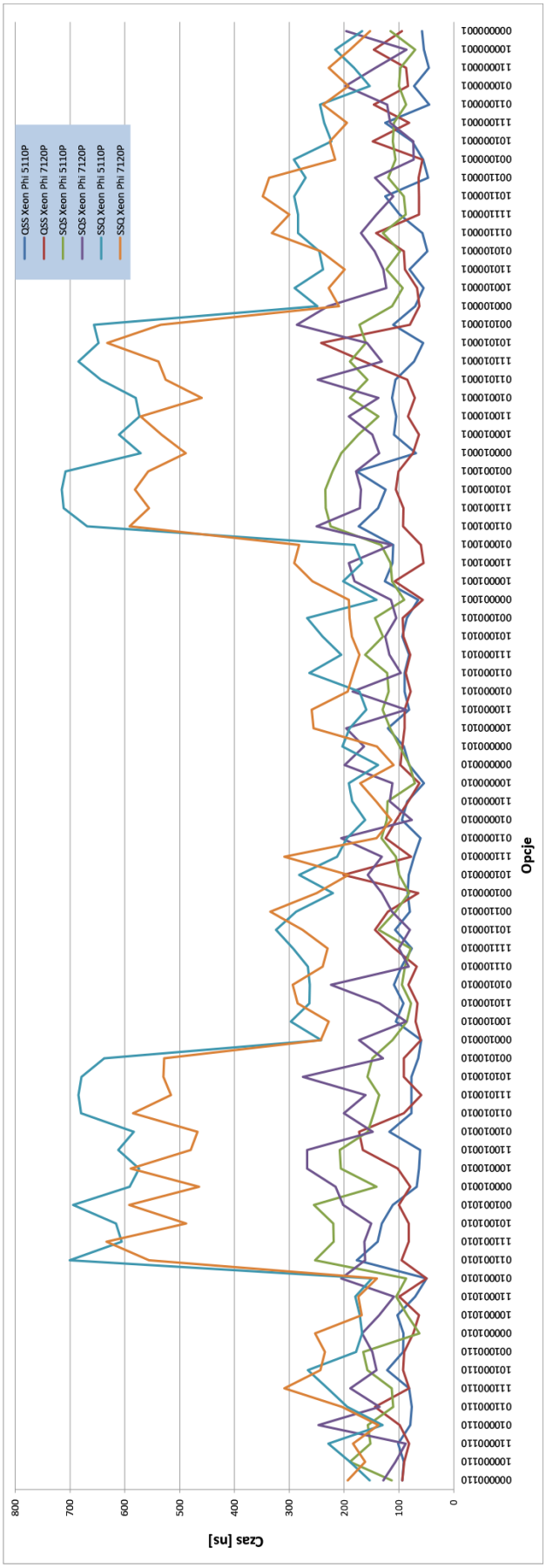
issona z elementami pryzmatycznymi (Rys. 5.15). Wyniki pozostałych zadań można znaleźć w aneksie C na końcu niniejszej pracy.

Porównując uzyskane wyniki z wynikami uzyskanymi na GPU oraz CPU możemy zauważyć że wykresy dla badanych koprocesorów są, tak jakby, złożeniem poprzednio uzyskanych charakterystyk. W skali mikro możemy zaobserwować duże fluktuacje wyniku charakterystyczne dla jednostek CPU. Za to w skali makro widzimy tą samą powtarzalność wyników co dla GPU. Dla elementów pryzmatycznych oraz algorytmu SSQ widzimy bardzo duże skoki wydajności związane z użyciem opcji COMPUTE_ALL_SHAPE_FUN_DER, która w tym przypadku prowadzi do redundantnych obliczeń. To samo powoduje opcja przechowywania tablicy z danymi geometrycznymi (GEO) w pamięci wspólnej, co również w przypadku wersji SSQ algorytmu powoduje redundantność pobrań. Ze względu na małą ilość pamięci cache L1, która w modelu OpenCL jest traktowana jako pamięć wspólna, w przypadku dużej redundantności obliczeń używane są wolniejsze poziomy pamięci.

Tabela 5.31: Zestawienie najlepszych wyników w *ns* dla poszczególnych wersji algorytmu całkowania numerycznego dla koprocesorów Xeon Phi

		Poisson		**konwekcja-dyfuzja**	
		czworościan	*pryzma*	*czworościan*	*pryzma*
5110P	**QSS**	17,746	**44,685**	27,157	**46,610**
	SQS	**14,397**	62,857	**24,956**	81,547
	SSQ	17,591	129,838	25,793	175,491
7120P	**QSS**	**22,986**	**49,870**	32,660	**54,367**
	SQS	23,085	73,167	**31,474**	76,783
	SSQ	24,962	109,415	31,526	171,484

Zestawienie najlepszych wyników dla badanych wersji OpenCL prezentuje tabela 5.31. Jak widać z tabeli, najlepsze wyniki uzyskano dla algorytmu SQS dla elementów czworościennych oraz QSS dla pryzmatycznych, co jest analogiczne do wyników



Rysunek 5.15: Automatyczny tuning parametryczny dla koprocesorów Intel Xeon Phi oraz zadania Poissona i elementów pryzmatycznych

Tabela 5.32: Zestawienie najlepszych kombinacji opcji dla poszczególnych wersji algorytmu całkowania numerycznego na Xeonie Phi

		Poisson		**konwekcja-dyfuzja**	
		czworościan	*pryzma*	*czworościan*	*pryzma*
5110P	**QSS**	0100100	**0110000**	1000010	**1101000**
	SQS	**0110100**	0000010	**0100100**	0110000
	SSQ	0010100	0100001	0100000	0100001
7120P	**QSS**	**1110100**	**0100010**	0110000	**0111000**
	SQS	0100000	0010000	**1100010**	0100000
	SSQ	0000010	0000000	1100100	0110000

uzyskanych na GPU oraz CPU. Kombinacje opcji z najlepszymi uzyskanymi wynikami prezentuje tabela 5.32.

Jak widać, opcje ciągłego zapisu oraz odczytu, nie odgrywają tu większej roli. Aż połowa opcji dla koporocesora Xeon Phi 7120P zakłada nie używanie pamięci wspólnej, co przy ograniczonej liczbie rejestrów dla tej architektury może być przyczyną chaotycznych i gorszych wyników niż teoretycznie wolniejszy Xeon Phi 5110P.

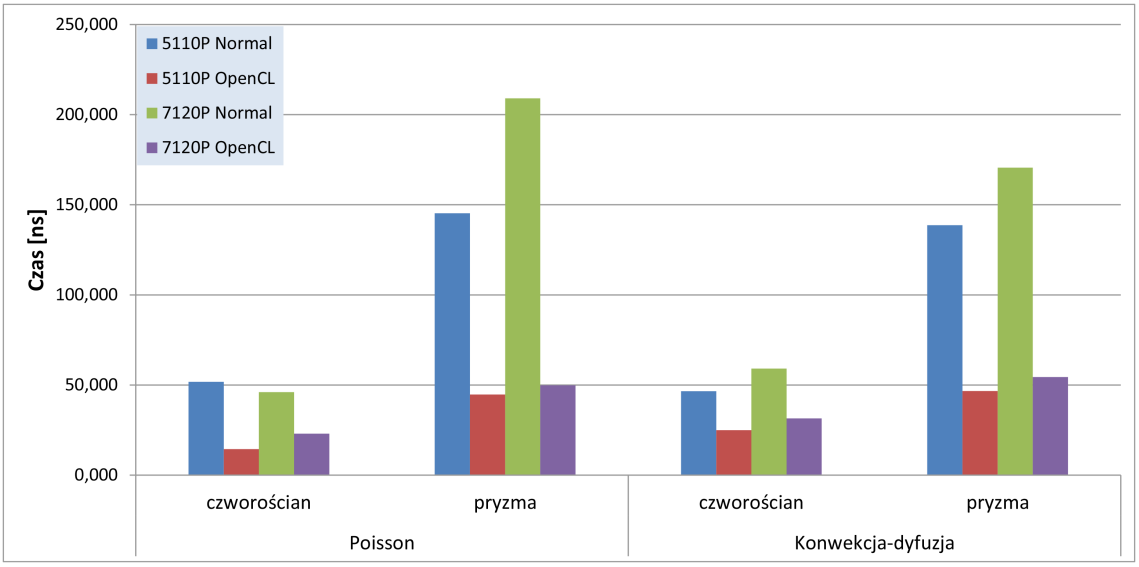
Jak widać w przypadku Xeona Phi o architekturze KNC, wyniki uzyskane w modelu OpenCL są lepsze niż te uzyskane z użyciem OpenMP oraz modelu Offload. Może to świadczyć o lepszym wykorzystaniu wektoryzacji i wektorowych dostępów do pamięci przez OpenCL, niż przez klasyczny sposób programowania z użyciem OpenMP. Uzyskany procent teoretycznej wydajności prezentuje tabela 5.33.

Jak można zauważyć uzyskano o wiele wyższy stopień wykorzystania badanych koprocesorów sięgający 57% dla najbardziej wymagającego zadania z dużą ilością redundantnych obliczeń. Jednakże należy zauważyć że dla najlepszych czasów uzyskany procent wydajności sięga jedynie 13,4% dla procesora 5110P oraz 8,9% dla 7120P.

Porównanie uzyskanych wyników prezentuje rysunek 5.16.

Tabela 5.33: Uzyskany procent wydajności teoretycznej dla badanych procesorów Xeon Phi oraz implementacji algorytmu w OpenCL

		Poisson		**konwekcja-dyfuzja**	
		czworościan	*pryzma*	*czworościan*	*pryzma*
5110P	*QSS*	9,84%	9,17%	9,28%	13,41%
	SQS	12,12%	22,34%	10,10%	19,86%
	SSQ	9,92%	57,33%	15,41%	47,38%
7120P	*QSS*	6,92%	6,32%	7,04%	8,85%
	SQS	6,89%	14,78%	7,30%	16,24%
	SSQ	6,37%	52,37%	9,70%	37,32%

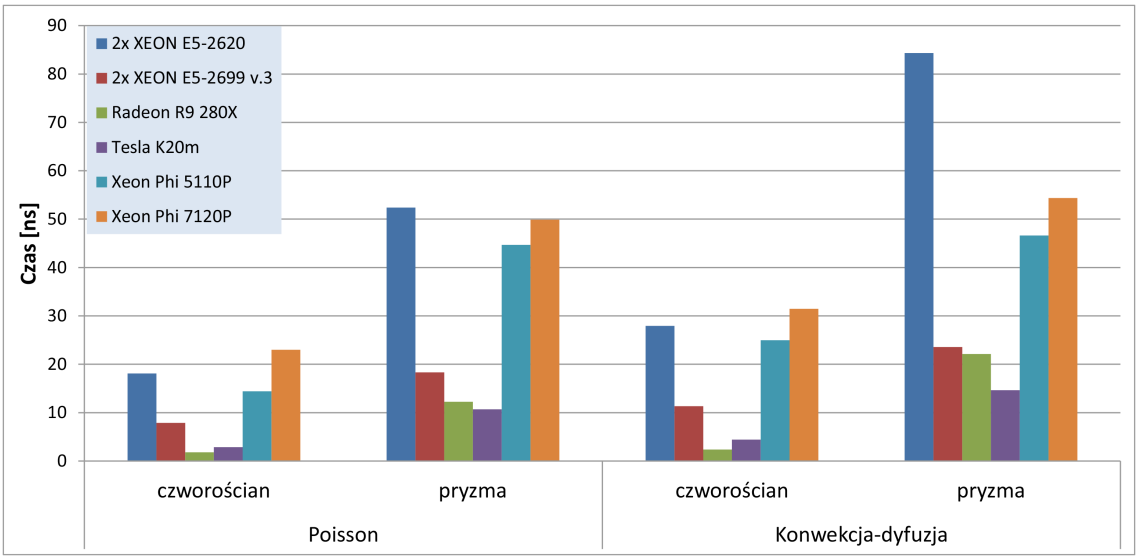


Rysunek 5.16: Porównanie wyników pomiędzy wersją OpenMP a OpenCL algorytmu całkowania numerycznego na koprocesorach Intel Xeon Phi

Uzyskane wyniki oraz przeprowadzone badania świadczą o tym, że w przypadku całkowania numerycznego, lepsze rezultaty uzyskujemy dla uruchamiania kodu dedykowanego dla GPU niż dla CPU.

5.4.2.3. Porównanie wyników

Rysunek 5.17 przedstawia wyniki wszystkich badanych typów urządzeń dla zadań liniowej aproksymacji standardowej.



Rysunek 5.17: Porównanie najlepszych wyników uzyskanych dla algorytmu całkowania numerycznego na procesorach CPU, akceleratorach GPU oraz koprocesorach Intel Xeon Phi

Jak można zauważyć, wydajność badanych koprocesorów jest porównywalna z procesorem o architekturze Sandy Bridge. Jest to zgodne z wynikami działania programów testujących zaprezentowanych w tabelach 5.4 oraz 5.27. Świadczy to o tym, że architektura Knights Corner jest już dosyć przestarzała i nie nadąża za nowymi procesorami opartymi o architekturę Haswell. Równocześnie należy zauważyć, że teoretycznie wolniejszy procesor Xeon Phi 5110P uzyskał wyższą wydajność niż szybszy 7120P. Może to świadczyć o tym, że dla algorytmu całkowania numerycznego nie jest koniecznym korzystanie ze wszystkich dostępnych wątków. Ograniczenie ilości wątków, mogłoby pozwolić na zwiększenie ilości dostępnych rejestrów wektorowych dla każdego z nich. Uzyskane wyniki pokazują dużą przydatność i dobre możliwości adaptacji opracowanego przez autora systemu automatycznego tuningu parametrycznego badanego algorytmu całkowania numerycznego.

5.4.2.4. Wnioski

Badane koprocesory Intel Xeon Phi są konstrukcjami o ciekawej budowie i różnorodnych metodach programowania. Jednakże sama ich architektura, oparta na starej architekturze Pentium 4 poszerzonej o szerokie rejestry wektorowe, nie wystarczyła do uzyskania wysokiej wydajności dla algorytmu całkowania numerycznego z liniową aproksymacją.

Badania przeprowadzone w ramach niniejszego rozdziału sprawdziły dogłębnie dwie główne zalety badanych koprocesorów czyli ich wydajność oraz prosty model programowania. W obu przypadkach autor musi jednak stwierdzić, iż programowanie tychże koprocesorów nie jest ani łatwe, ani nie prowadzi do nadzwyczajnej wydajności. Jest to zgodne ze wspomnianymi wcześniej badaniami innych autorów. Przeprowadzone testy wskazują, że niniejsza konstrukcja wydaje się już rozwiązaniem przestarzałym i zdolnym do rywalizacji jedynie z porównywalnymi czasowo architekturami CPU. Głównymi czynnikami ograniczającymi wydajność algorytmu całkowania numerycznego jest przestarzała architektura oraz mała ilość rejestrów dostępna na wątek. Mimo tego, należy zauważyć, że dla procesora 5110P uzyskano lepsze wyniki niż dla procesora o architekturze Sandy Bridge pochodzącej z tego samego okresu. Daje to nadzieję, że poprawione wersje koprocesorów Intel Xeon Phi o architekturze *Knights Landing* mogą rzeczywiście być już realną alternatywą dla odpowiadających sobie architektur CPU.

5.5. AMD APU

5.5.1. Metodologia

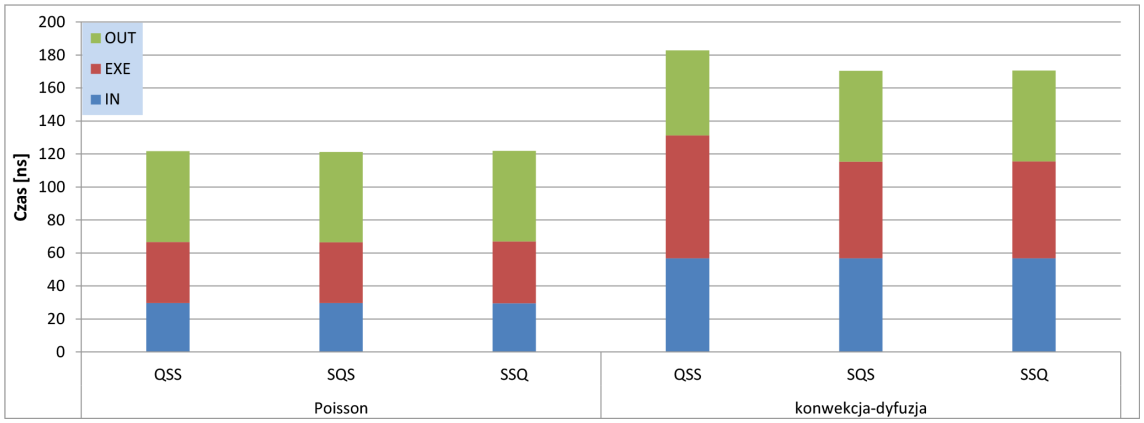
Procesor AMD APU A10-7850, wykorzystywany w badaniach, jest jednostką hybrydową stosowaną głównie w konstrukcjach przenośnych i energooszczędnych, takich jak laptopy lub komputery biurowe. Należy zauważyć, że jego możliwości obliczeniowe są ograniczone. Z tego względu, został on potraktowany jako zapowiedź przyszłych konstrukcji hybrydowych, eliminujących problem przesyłu danych między pamięcią akceleratora i hosta. W swoich badaniach autor skupił się więc na problemie przesyłu danych oraz mechanizmach go wspomagających, zaimplementowanych w badanym procesorze. W celu dokonania badań, skorzystano z systemu automatycznego tuningu dla zadań z liniową aproksymacją standardową, opisanego we wcześniejszych rozdziałach. Mimo tego że jednostka APU korzysta w pełni z możliwości architektury HSA, dla systemu operacyjnego jest widoczna jako dwie osobne jednostki GPU oraz CPU. Dzięki temu, korzystając z klasycznej implementacji algorytmu całkowania numerycznego w OpenCL, możliwym stało się przetestowanie różnic pomiędzy standardowym mechanizmem przesyłu danych na akcelerator, a nowym mechanizmem współdzielonego bufora danych (SVM) omówionym w rozdziale 2.5.

5.5.2. Wyniki

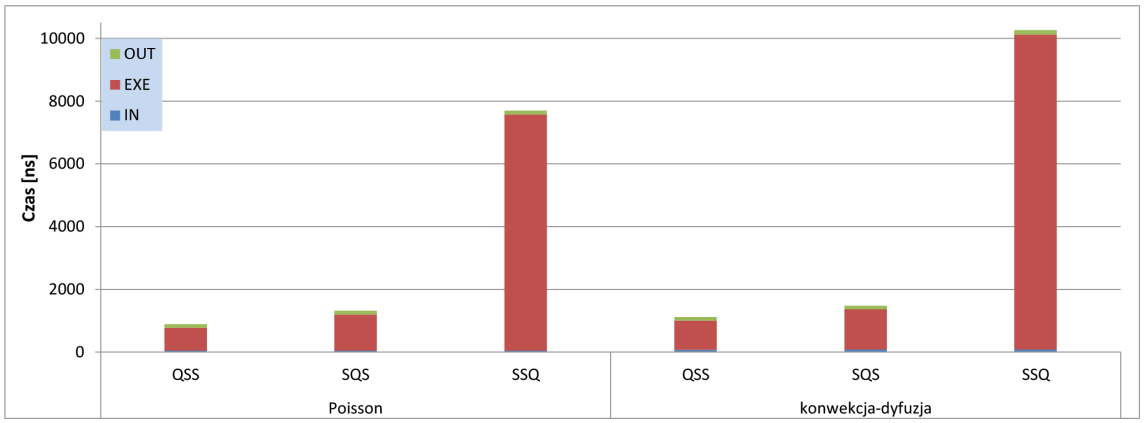
Uzyskane wyniki klasycznej implementacji OpenCL obrazuje rysunek 5.18 dla elementów czworościennych oraz 5.19 dla elementów pryzmatycznych. Na rysunkach oznaczono poszczególne etapy wykonania: *IN* - czas przesłania danych wejściowych, *EXE* - czas obliczeń, *OUT* - czas przesłania danych wyjściowych.

Jak widać na wykresach, w przypadku elementów czworościennych, duża część czasu przeznaczona jest na przesłanie danych pomiędzy obszarem pamięci akceleratora GPU a obszarem pamięci CPU. W przypadku elementów pryzmatycznych, czas obliczeń zdecydowanie dominuje czasy przesłania. Z wykresów 5.20 oraz 5.21 wynika że dla elementów czworościennych, czas przesłania danych sięga 60% całkowitego czasu wykonania, a dla elementów pryzmatycznych waha się od 1% do 18%.

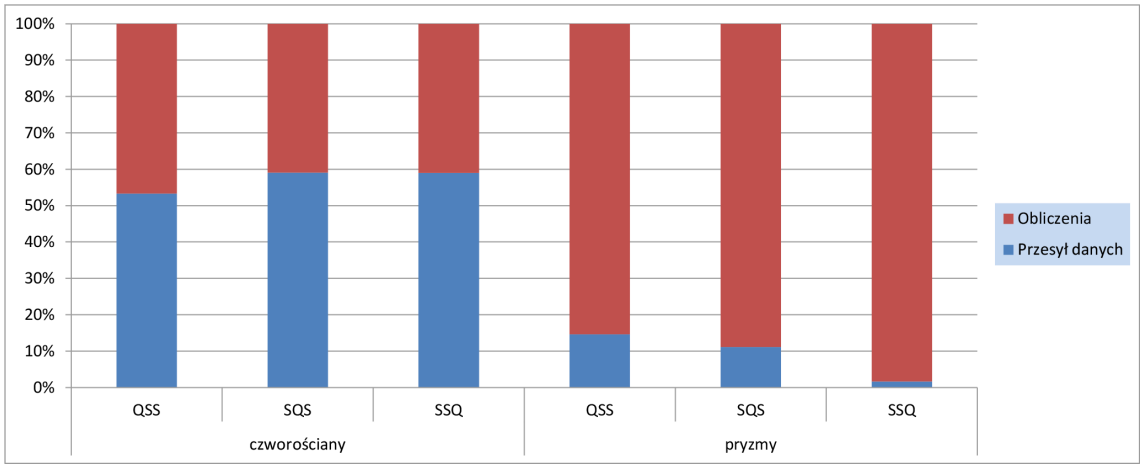
W celu sprawdzenia wydajności przesłania, obliczono przepustowość dla poszczególnych wersji algorytmu (Tab. 5.34). W przypadku jednostki zintegrowanej, podana przepustowość dotyczy odczytu danych z jednego obszaru pamięci RAM, a następnie zapisu w drugi. W związku z czym można szacować teoretyczną prędkość tego



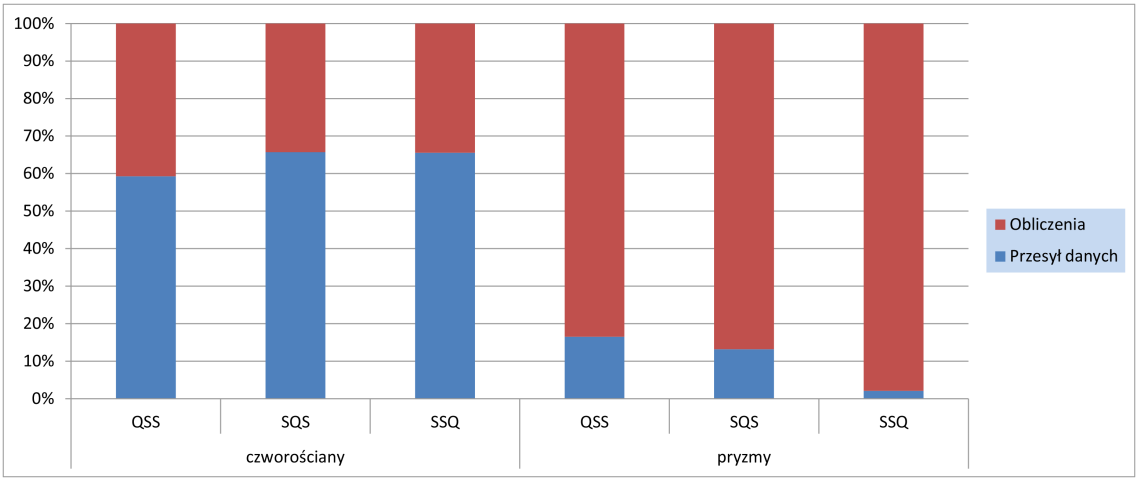
Rysunek 5.18: Zestawienie najlepszych wyników uzyskanych dla klasycznej implementacji OpenCL dla algorytmu całkowania numerycznego i elementów czworościennych na procesorze APU



Rysunek 5.19: Zestawienie najlepszych wyników uzyskanych dla klasycznej implementacji OpenCL dla algorytmu całkowania numerycznego i elementów pryzmatycznych na procesorze APU



Rysunek 5.20: Procentowy udział transferu danych oraz obliczeń dla klasycznej implementacji OpenCL dla algorytmu całkowania numerycznego i zadania Poissona na procesorze APU



Rysunek 5.21: Procentowy udział transferu danych oraz obliczeń dla klasycznej implementacji OpenCL algorytmu całkowania numerycznego i zadania konwekcji-dyfuzji na procesorze APU

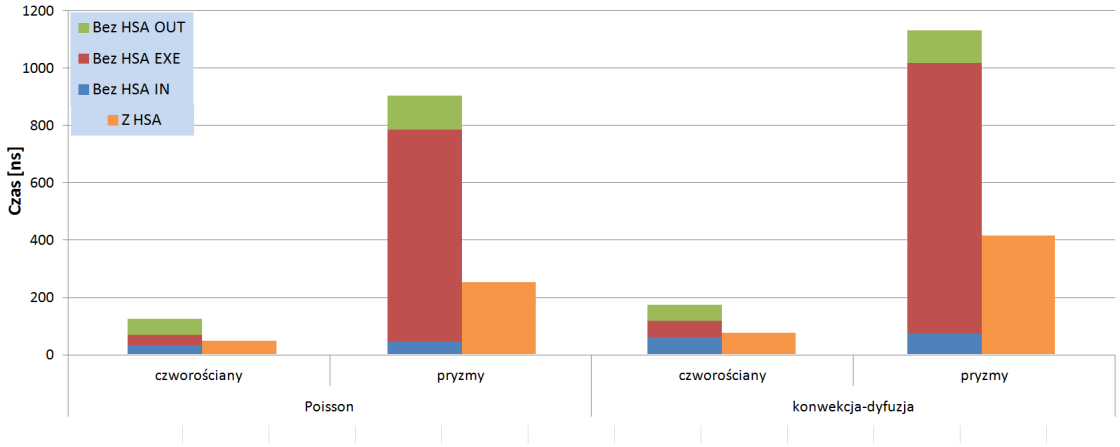
Tabela 5.34: Rozmiary danych dla pojedynczego elementu oraz uzyskane przepustowości dla procesora APU i klasycznej implementacji algorytmu całkowania numerycznego

	Poisson				**konwekcja-dyfuzja**			
	czworościany		*pryzmy*		*czworościany*		*pryzmy*	
	WE	**WY**	**WE**	**WY**	**WE**	**WY**	**WE**	**WY**
Rozmiar danych [bajty]								
	128	160	192	336	256	160	304	336
Przepustowość [GB/s]								
QSS	4,30	2,90	4,34	2,90	4,35	2,94	4,31	2,93
SQS	4,31	2,93	4,34	2,88	4,50	2,90	3,75	2,94
SSQ	4,33	2,92	4,34	2,70	4,51	2,90	3,74	2,49

interfejsu na 5,3 GB/s, gdyż badany system jest wyposażony w pamięć DDR3 o częstotliwości taktowania 1333 MHz i teoretycznej przepustowości 10,6 GB/s.

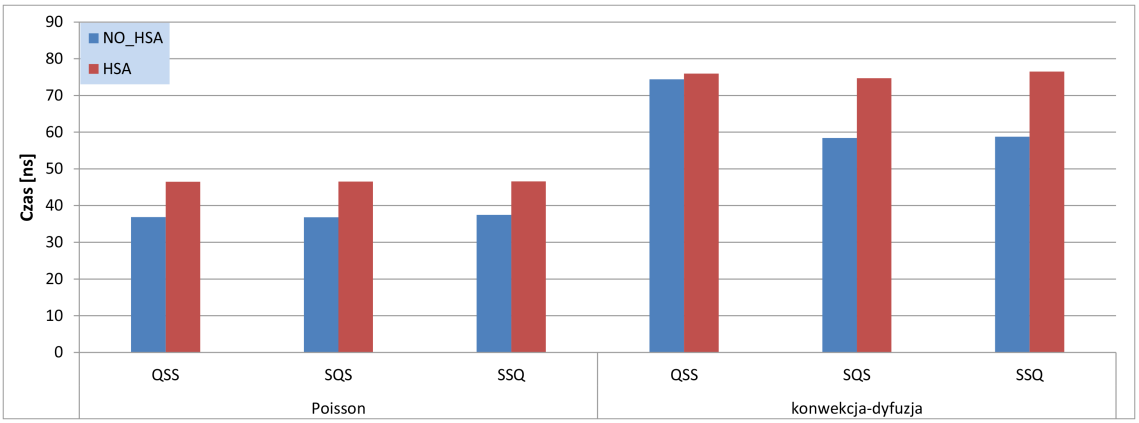
Jak można zauważyć z tabeli, w przypadku odczytu danych udało się uzyskać prawie 82% przepustowości, a w przypadku zapisu wartość ta oscylowała w granicach 50%.

Aby sprawdzić główne zalety architektury HSA, dokonano zmian w implementacji algorytmu, wprowadzając jeden obszar danych do komunikacji między jednostkami GPU oraz CPU (Shared Virtual Memory). Dzięki temu całkowicie wyeliminowano problem przesyłu danych. Porównanie obu implementacji dla najlepszych uzyskanych wyników obrazuje rysunek 5.22. Jak widać z wykresu, w przypadku elementów czworościennych i zadania Poissona uzyskano 2,6 krotne przyśpieszenie a w przypadku zadania konwekcji-dyfuzji 2,3 krotne. W przypadku pryzm uzyskano imponujące przyśpieszenie 3,5 raza dla zadania Poissona oraz 2,7 raza dla konwekcji-dyfuzji.



Rysunek 5.22: Porównanie najlepszych wyników między klasyczną implementacją OpenCL z przesyłem danych a implementacją z użyciem wspólnej pamięci HSA

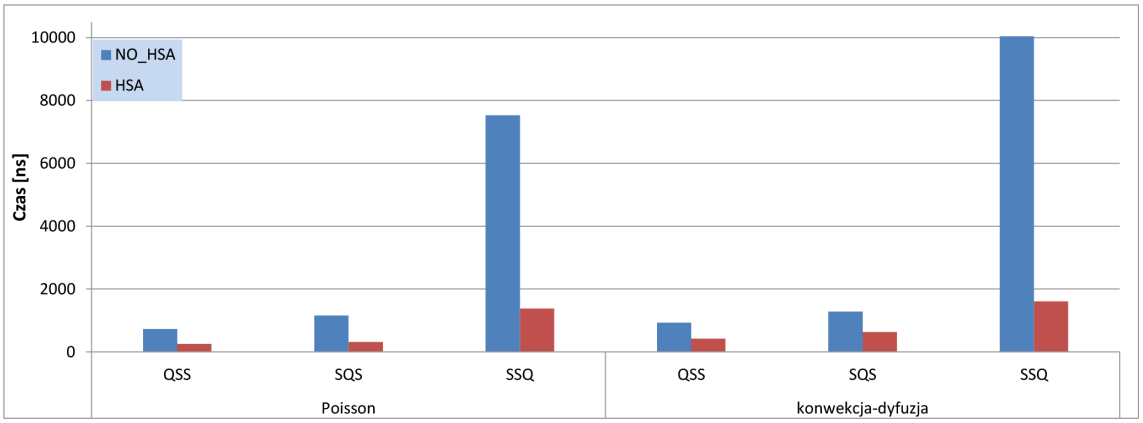
Co ciekawe, na wartość przyśpieszenia składa się nie tylko eliminacja przesłania ale również zmiana czasu wykonania. Dla elementów czworościennych wykonanie z użyciem wspólnego bufora SVM charakteryzuje się ok. 30% spadkiem wydajności (Rys. 5.23). W przypadku elementów pryzmatycznych sytuacja się odwraca i wykonanie HSA jest od 50% do aż 83% szybsze niż w przypadku klasycznej implementacji (Rys. 5.24).



Rysunek 5.23: Porównanie czasu obliczeń pomiędzy implementacją OpenCL z przesyłem danych (NO_HSA) a implementacją z użyciem wspólnej pamięci (HSA) dla elementów czworościennych

Różnice te świadczą o istnieniu lekkiego narzutu na komunikację z buforem SVM, który jest widoczny w przypadku elementów czworościennych o małej ilości obliczeń, a w przypadku bardziej wymagających obliczeniowo elementów pryzmatycznych jest ukrywany przez obliczenia.

Podsumowując, należy zauważyć że stosowanie mechanizmów architektury HSA zaimplementowanych w procesorze AMD APU A10-Kavieri przynosi bardzo obiecu-

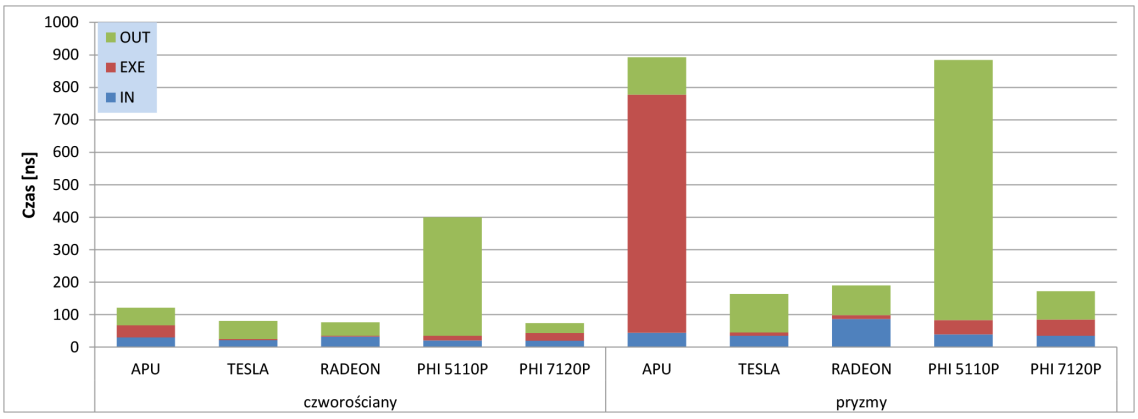


Rysunek 5.24: Porównanie czasu obliczeń pomiędzy implementacją OpenCL z przesyłem danych (NO_HSA) a implementacją z użyciem wspólnej pamięci (HSA) dla elementów pryzmatycznych

jące rezultaty. W celu oceny wpływu tejże architektury na akceleratory w większej mocy obliczeniowej, przebadano transport danych na tych architekturach. Pozwoli to oszacować ewentualny zysk, jaki byłby możliwy przy stosowaniu założeń HSA do tego typu akceleratorów.

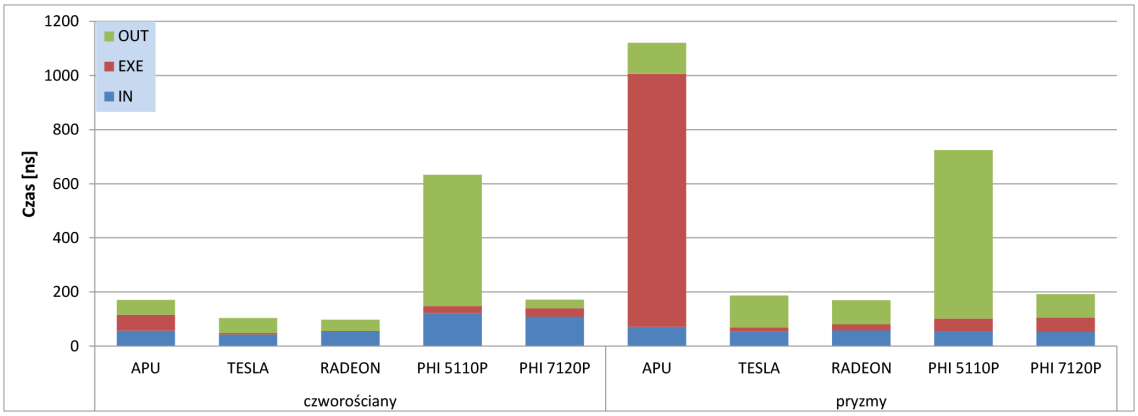
5.5.3. Porównanie wyników

Na rysunkach 5.25 oraz 5.26 przedstawiono wyniki działania algorytmu całkowania numerycznego wraz z zaznaczonym czasem transferu danych.

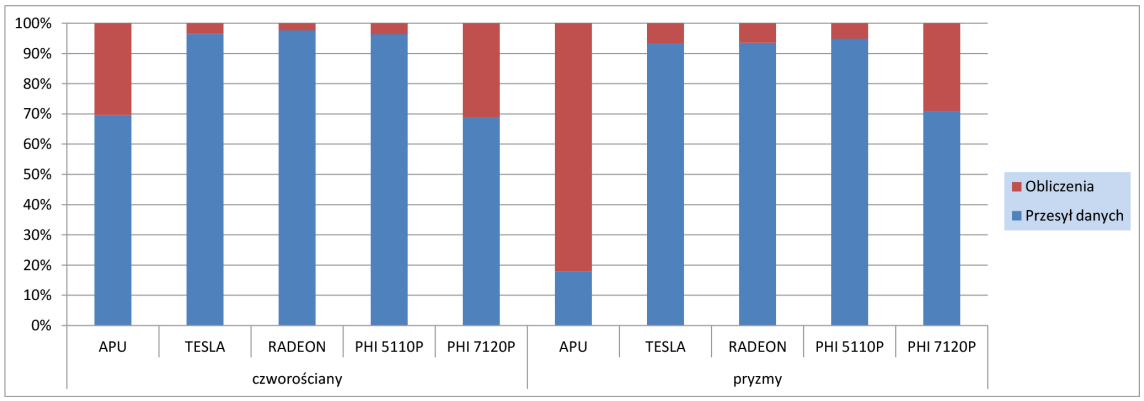


Rysunek 5.25: Porównanie czasu wykonania całego algorytmu wraz z transferem danych dla zadania Poissona i badanych akceleratorów

Jak można zauważyć, dla architektur o bardzo dużej szybkości obliczeń, czas transferu zdecydowanie dominuje całkowity czas wykonania. W przypadku akceleratorów GPU ponad 95% czasu wykonania algorytmu zajmuje przesył danych (Rys. 5.27, 5.28).



Rysunek 5.26: Porównanie czasu wykonania całego algorytmu wraz z transferem danych dla zadania konwekcji-dyfuzji i badanych akceleratorów

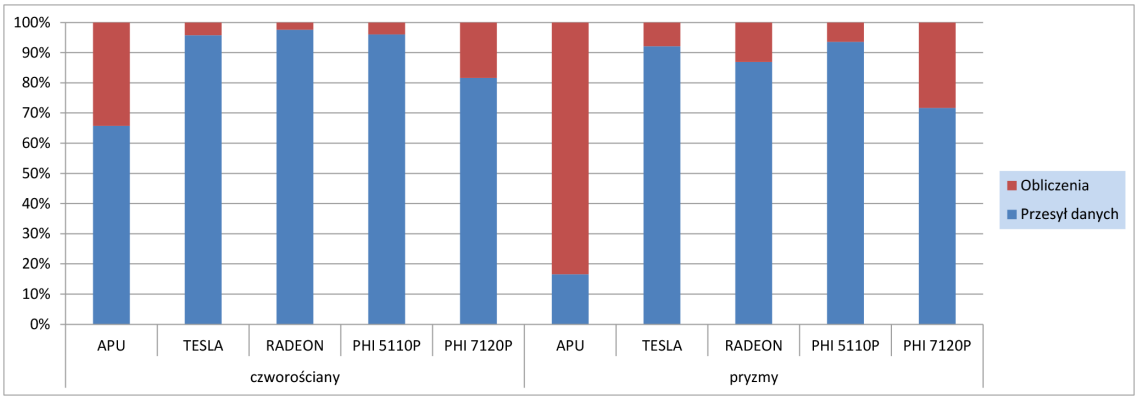


Rysunek 5.27: Porównanie stosunku czasu przesyłu danych do czasu obliczeń dla zadania Poissona i badanych akceleratorów

Większość z badanych akceleratorów korzysta z łącza PCI-Express 2.0 x16 o teoretycznej przepustowości 8 GB/s. Jedynie akcelerator Xeon Phi 7120P korzysta z łącza PCI-Express 3.0 x8 o przepustowości teoretycznej 7,88 GB/s. Osiągnięte transfery przedstawia tabela 5.35. Jak widać z tabeli, dla odczytu danych uzyskano wysokie przepustowości wahające się od 67% do 82%. W przypadku zapisu, wartości nie są już takie imponujące. Bardzo dziwny wynik koprocesora 5110P może wskazywać na kłopoty ze sterownikiem albo z samym sprzętem. Co ciekawe, koprocesor ten jest umieszczony w serwerze o identycznej konfiguracji sprzętowej co badana karta Tesla K20m, której prędkości zapisu są ponad dwukrotnie większe.

Jako podsumowanie należy porównać wyniki uzyskane na procesorach ogólnego przeznaczenia z wynikami z akceleratorów wraz z czasami transferów danych (Rys. 5.29).

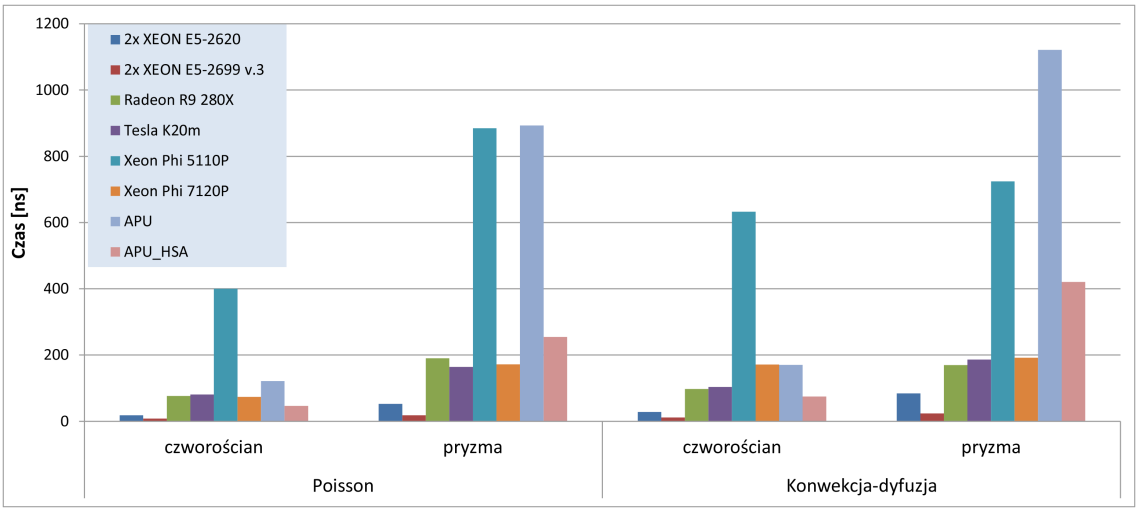
Jak widać z wykresu, aktualne akceleratory zewnętrzne, mimo ogromnej mocy obliczeniowej oraz bardzo dobrych czasów przetwarzania, nie dają sobie niestety



Rysunek 5.28: Porównanie stosunku czasu przesyłu danych do czasu obliczeń dla zadania konwekcji-dyfuzji i badanych akceleratorów

Tabela 5.35: Uzyskane przepustowości maksymalne dla przesyłu danych w algorytmie całkowania numerycznego - w nawiasach przedstawiono uzyskany procent wartości teoretycznej

	Przepustowość uzyskana [GB/s]	
	WE	**WY**
APU(bez HSA)	4,34 (82%)	2,94 (55%)
TESLA	5,76 (72%)	2,83 (35%)
RADEON	5,32 (67%)	3,76 (47%)
PHI 5110P	5,52 (69%)	1,34 (17%)
PHI 7120P	5,92 (75%)	3,90 (50%)



Rysunek 5.29: Porównanie czasów wykonania całego algorytmu całkowania numerycznego na badanych architekturach

rady z problemem przesyłu danych. Powoduje to, że nawet starszy technologicznie

procesor o architekturze Sandy Bridge, okazuje się być kilkukrotnie szybszy niż najwydajniejsze z akceleratorów.

5.5.4. Wnioski

Tak jak wspomniano we wcześniejszych rozważaniach, problem przesyłu danych na zewnętrzne akceleratory powoduje, iż zadania takie jak całkowanie numeryczne, nie są w stanie uzyskać dużego przyśpieszenia czasu wykonania. Jednakże, jak pokazują uzyskane wyniki, sam czas obliczeń może być zoptymalizowany i wydatnie przyśpieszony na tego typu architekturach. Wraz z opracowaniem standardu Heterogenous System Architecture pojawia się nadzieja, że przyszłe jednostki wspomagające obliczenia będą pozbawione tego jedynego minusu w postaci wolnego interfejsu łączącego akcelerator z pamięcią systemu gospodarza.

5.6. Podsumowanie

W niniejszym rozdziale omówione zostały rożne sposoby implementacji algorytmu całkowania numerycznego na procesory o zróżnicowanych architekturach oraz korzystające z wielopoziomowej hierarchii pamięci. Testowany algorytm, cechuje duża złożoność związana z jego uzależnieniem od rodzaju elementu na jaki zostanie podzielona domena obliczeniowa MES, rozwiązywanego problemu oraz typu aproksymacji. Przetestowany zakres zadań i aproksymacji sprawia, iż badane architektury zostały przetestowane kompleksowo pod wieloma kątami, a uzyskane wyniki mogą stanowić bazę do implementacji szerokiej gamy algorytmów numerycznych na te lub podobne architektury. Dzięki pracom omówionym w niniejszym rozdziale, możliwym stało się zrealizowanie celu pracy, czyli znalezienie sposobu na efektywną realizację algorytmu całkowania numerycznego w MES na nowoczesnych architekturach komputerowych.

Rozdział 6

Zakończenie rozprawy

Główny cel pracy stanowiło znalezienie odpowiedzi na pytanie w jaki sposób efektywnie realizować tworzenie macierzy sztywności w równoległych symulacjach Metody Elementów Skończonych z wykorzystaniem wielopoziomowej hierarchii pamięci i architektur masowo wielordzeniowych. Dzięki temu, możliwym jest uzyskanie wskazówek dotyczących projektowania podobnych algorytmów, jak i istotnych wskazówek dotyczących wyboru sprzętu odpowiedniego do tego typu zagadnień.

W podrozdziale 5.1, autor opisał implementację zadania całkowania numerycznego dla nieciągłej aproksymacji Galerkina i problemu Poissona na procesorze hybrydowym PowerXCell 8i. Uzyskane wyniki pokazały bardzo duży potencjał badanej architektury, oraz jej spore możliwości w zakresie przyśpieszenia wykonania algorytmu całkowania numerycznego, szczególnie dla wyższych stopni aproksymacji. Wyniki zaprezentowane w tym rozdziale były przedłużeniem prac opublikowanych w [66] i [67] oraz prezentowanych na międzynarodowych konferencjach *Parallel Processing and Applied Mathematics* (Wrocław, 2009) oraz *Higher Order Finite Element and Isogeometric Methods* (Kraków, 2011). Kluczowym wnioskiem dotyczącym programowania na architekturę Cell/BE jest konieczność wykorzystania dwóch potoków przetwarzania, operacji FMA oraz rejestrów wektorowych.

W pierwszej części podrozdziału 5.2, autor skupił się na opracowaniu wydajnej implementacji algorytmu całkowania numerycznego dla liniowej aproksymacji standardowej oraz problemów Poissona i konwekcji-dyfuzji, na procesory ogólnego przeznaczenia (CPU). Przetestowano również różnice pomiędzy stosowanymi różnymi typami elementów na które został podzielony obszar obliczeniowy. W trakcie badań, autor przetestował różne techniki optymalizacyjne i uzyskał bardzo dobre wyniki przyśpieszeń i wydajności dla badanych wersji algorytmu. W wyniku badań, autor wysnuł wniosek, że zadania bardziej wymagające obliczeniowo i o skomplikowanej geometrii, uzyskują lepsze wartości przyśpieszeń. Badania zaprezentowane w podrozdziale 5.2 były częścią badań wykonanych w ramach przyznanej nagrody w konkursie pt. *Najciekawsze Wykorzystanie Architektury Intel MIC Oraz Wielor-*

dzeniowych Procesorów Ogólnego Przeznaczenia - w ramach projektu pn. *"Laboratorium pilotażowe systemów masywnie wielordzeniowych (MICLAB)"* prowadzonego na Politechnice Częstochowskiej. W drugiej części podrozdziału autor rozszerzył opracowaną wcześniej implementację o rozwiązanie problemu konwekcji-dyfuzji dla nieciągłej aproksymacji Galerkina. Zgodnie ze wcześniejszym wnioskiem, okazało się, że zadanie bardziej wymagające obliczeniowo jest w stanie w większym stopniu wykorzystać potencjał nowoczesnych procesorów ogólnego przeznaczenia i osiągnąć bardzo dobre wyniki przyśpieszeń. Kluczową sprawą wydaje się w tym przypadku odpowiednie wykorzystanie wektoryzacji obliczeń oraz wykorzystanie dużej ilości cache dla nowszych architektur procesorów.

W podrozdziale 5.3, autor opisał opracowany przez siebie system automatycznego tuningu parametrycznego algorytmu całkowania numerycznego, pozwalający na optymalne dobranie parametrów do badanego sprzętu. W sekcji tej, zbadano przydatność akceleratorów opartych na budowie kart graficznych, do przyśpieszania testowanego algorytmu. Wykazano wysoką wydajność badanego sprzętu i jego dużą użyteczność dla przyśpieszenia badanego algorytmu. Autor dokonał dokładnej analizy badanego sprzętu, pod kątem wykorzystania wielopoziomowej struktury pamięci oraz jednostek przetwarzających w modelu SIMD. W niniejszym fragmencie, autor zauważył istotny problem przesyłu danych na akcelerator, który opisał dokładnie w podrozdziale 5.5. Dodatkowo, autor dokonał porównania wyników uzyskanych dla architektur opartych o budowę GPU z wynikami dla architektur standardowych CPU. Pozwoliło to wysnuć wniosek, że współczesne procesory CPU, zbliżają się zarówno pod względem architektury jak i wydajności do jednostek GPU, głównie poprzez stosowanie coraz większej ilości dostępnych rdzeni obliczeniowych oraz coraz szerszych jednostek wektorowych. Badania z sekcji 5.3 były rozwinięciem badań zaprezentowanych w [7], [5] i [6]. Dla tego typu architektur, bardzo dużą rolę odgrywa wielopoziomowa struktura pamięci oraz stosunkowo proste w budowie jednostki SIMD - wymusza to maksymalizację liczby obliczeń w stosunku do liczby pobrań z pamięci i może być kluczowym czynnikiem decydującym o wyborze algorytmu do uruchomienia na procesorze graficznym.

W podrozdziale 5.4 autor przebadał koprocesor Intel Xeon Phi (w dwóch wersjach) pod względem jego użyteczności dla badanego algorytmu całkowania numerycznego. W wyniku badań autora dla aproksymacji liniowej okazało się, że mimo opracowania optymalnej wersji algorytmu dla procesora Intel Xeon, opisanej we wcześniejszym podrozdziale, architektura *Knights Corner* o wiele lepiej sobie radzi z użyciem wersji opracowanej dla GPU. Jak zauważył autor, osiągniecie wysokiej wydajności na tych koporocesorach wymaga niestandardowych optymalizacji

programistycznych i może być uważane za równie trudne jak programowanie GPU. Wyniki zaprezentowane w tym podrozdziale są przedłużeniem badań zaprezentowanych w cytowanych wcześniej pracach [7], [5], [6] oraz [65]. Zostały one zaprezentowane przez autora na konferencji pn. *Federated Conference on Computer Science and Information Systems* (Warszawa, 2014). Były one także częścią badań wykonanych w ramach wspomnianego wcześniej grantu MICLAB.

W podrozdziale 5.5, autor skupił się na problemie przesyłu danych pomiędzy pamięcią akceleratora a pamięcią systemu gospodarza na którym ten akcelerator jest zainstalowany. Autor opisał jednostkę Accelerated Procesing Unit jako wcielającą w życie paradygmat Heterogenous System Architecture i przetestował jej główną zaletę, jaką jest użycie jednej przestrzeni adresowej zarówno dla procesora ogólnego przeznaczenia jak i specjalistycznego urządzenia (w tym przypadku GPU). Uzyskane przez autora wyniki świadczą o ogromnym przyroście wydajności dzięki stosowaniu tego typu rozwiązania. W niniejszym podrozdziale autor porównuje dotychczas uzyskane wyniki, uwzględniając czasy przesyłu po wolnych interfejsach łączących badane akceleratory z pamięcią RAM systemu gospodarza. Pokazuje to istotność problemu przesyłu danych w algorytmie całkowania numerycznego, równocześnie implikując potencjalne zyski z zastosowania paradygmatu HSA w budowie architektur stricte obliczeniowych. Wyniki zaprezentowane w tej sekcji są rozwinięciem badań autora zaprezentowanych w [68] oraz prezentowanych na konferencji *KomPlasTech* (Krynica-Zdrój, 2015).

Wszystkie zaprezentowane badania są efektem wieloletniej pracy autora nad przetestowaniem różnych architektur oraz sposobu optymalizacji badanego algorytmu całkowania numerycznego w MES. Z badań autora wynika, że najbardziej efektywnym sposobem symulacji, działającej na przeróżnych architekturach, jest zastosowanie tuningu parametrycznego, dostosowującego badany algorytm do sprzętu. Pozwala to na wydajne wykonanie algorytmu na praktycznie dowolnej architekturze i jej wykorzystanie w możliwie maksymalnym stopniu. Badania autora pokazują, iż algorytmy takie jak całkowanie numeryczne, można w sposób efektywny zrównoleglić i zwektoryzować, a ich optymalizacja powinna być dostosowana do używanej geometrii elementów, typu aproksymacji oraz rozwiązanego zadania. W przypadku zadań o wyższych stopniach aproksymacji i nieliniowej geometrii elementów, zadanie całkowania numerycznego może stanowić istotny element całkowitego czasu wykonania zadania MES. Z tego względu, jego odpowiedni sposób optymalizacji jest niezwykle istotny. W tego typu przypadkach należy brać pod uwagę dostępne zasoby na danej architekturze, obejmujące szybkie poziomy pamięci, rejestry wektorowe i inne możliwości sprzętowe wspomagające wykonanie. W przypadku liniowej

aproksymacji standardowej, kluczowym elementem wydaje się być prędkość interfejsu łączącego jednostkę obliczeniową z główną pamięcią systemu. Zgodnie z przewidywaniami autora, obecne trendy w budowie akceleratorów do zadań obliczeń wysokiej wydajności, skupiają się na rozwiązaniu tego problemu. Firma Nvidia, wraz z wprowadzeniem nowej architektury pn. *Pascal*, wprowadziła superszybki interfejs *NVlink* o teoretycznej przepustowości 80 GB/s [93]. Intel w swoich nowych koprocesorach Intel Xeon Phi o mikroarchitekturze *Knights Landing*, wprowadza opcję uruchamiania tego koprocesora jako standardowego CPU zarządzającego systemem, co całkowicie eliminuje problem przesyłu danych [113]. Dzięki badaniom autora, możliwe jest przewidzenie, że po likwidacji głównej bariery wydajnościowej, współczesne akceleratory będą niezwykle przydatne do przyśpieszania, innych niż finalne rozwiązanie układu równań, procedur Metody Elementów Skończonych. Równocześnie, jak zauważył autor, współczesne procesory swoją architekturą coraz bardziej zbliżają się do jednostek masowo-wielordzeniowych i wektorowych, co sprawia, że możliwym jest uzyskanie wysokiej wydajności bez korzystania z zewnętrznych akceleratorów. Niezbędna jest tu jedynie odpowiednia organizacja kodu, w celu ułatwienia wektoryzacji i zrównoleglenia. Dzięki możliwościom współczesnych kompilatorów, pozwala to na efektywne wykorzystanie dostępnych zasobów sprzętowych i odpowiednią optymalizację wykonania.

Równocześnie praca autora dostarczyła wskazówek, jakie elementy nowoczesnych architektur komputerowych są najbardziej pożądane przy rozwiązywaniu tego typu problemów. Autor zauważył, iż odpowiednio duża liczba rejestrów wraz z efektywnym sposobem przesyłania danych z pamięci RAM systemu, daje wyniki w postaci największego przyśpieszenia. Daje to nadzieję, że wraz z nowymi architekturami, opracowany system tuningu parametrycznego badanego algorytmu, pozwoli na uzyskanie w sposób naturalny i całkowicie automatyczny, jak najwyższej możliwej wydajności.

Podsumowując, problem odpowiedniego odwzorowania algorytmu całkowania numerycznego w Metodzie Elementów Skończonych na współczesne architektury obliczeniowe, jest problemem niezwykle złożonym. Autor wyraża nadzieję, że dzięki badaniom przeprowadzonym w ramach tej pracy oraz uzyskanym wskazówkom, projektowanie tego typu efektywnych algorytmów będzie zadaniem łatwiejszym.

Bibliografia

[1] AMD. AMD Graphics Cores Next (GCN) architecture. Tech. rep., Advanced Micro Devices Inc., 2012.

[2] AMD. *AMD Accelerated Parallel Processing. OpenCL Programming Guide*, 2013. rev. 2.7.

[3] Arbenz, P., van Lenthe, G., Mennel, U., Muller, R., and Sala, M. A scalable multilevel preconditioner for matrix-free μ-finite element analysis of human bone structures. *International Journal for Numerical Methods in Engineering 73*, 7 (2008), 927–947.

[4] Baboulin, M., Buttari, A., Dongarra, J., Kurzak, J., Langou, J., Langou, J., Luszczek, P., and Tomov, S. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications 180*, 12 (2009), 2526 – 2533.

[5] Banaś, K., and Krużel, F. Opencl performance portability for Xeon Phi coprocessor and NVIDIA GPUs: A case study of finite element numerical integration. In *Euro-Par 2014: Parallel Processing Workshops*, vol. 8806 of *Lecture Notes in Computer Science*. Springer International Publishing, 2014, pp. 158–169.

[6] Banaś, K., Krużel, F., and Bielański, J. Finite element numerical integration for first order approximations on multi- and many-core architectures. *Computer Methods in Applied Mechanics and Engineering 305* (2016), 827 – 848.

[7] Banaś, K., and Krużel, F. Comparison of Xeon Phi and Kepler GPU performance for finite element numerical integration. In *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS), 2014 IEEE Intl Conf on* (Aug 2014), pp. 145–148.

[8] Becker, E., Carey, G., and Oden, J. *Finite Elements. An Introduction*. Prentice Hall, Englewood Cliffs, 1981.

[9] Buatois, L., Caumon, G., and Levy, B. Concurrent number cruncher: A GPU implementation of a general sparse linear solver. *Int. J. Parallel Emerg. Distrib. Syst. 24*, 3 (June 2009), 205–223.

[10] Cai, X.-C., Gropp, D., Keyes, E., Melvin, G., and Young, P. Parallel Newton-Krylov-Schwarz algorithms for the transonic full potential equation. Report TR 96-39, ICASE, 1996.

[11] Cai, Y., Li, G., and Wang, H. A parallel node-based solution scheme for implicit finite element method using GPU. *Procedia Engineering 61*, 0 (2013), 318 – 324. 25th International Conference on Parallel Computational Fluid Dynamics.

[12] Cecka, C., Lew, A. J., and Darve, E. Application of assembly of finite element methods on graphics processors for real-time elastodynamics. In *GPU Computing Gems. Jade edition*. Morgan Kaufmann, 2011, pp. 187–205.

[13] Cecka, C., Lew, A. J., and Darve, E. Assembly of finite element methods on graphics processors. *International Journal for Numerical Methods in Engineering 85*, 5 (2011), 640–669.

[14] Ciarlet, P. *The Finite Element Method for Elliptic Problems*. North-Holland, Amsterdam, 1978.

[15] Cockburn, B., Karniadakis, G., and Shu, C., Eds. *Discontinuous Galerkin Methods: Theory, Computation and Applications*, vol. 11 of *Lecture Notes in Computational Science and Engineering*. Springer, Berlin, 2000.

[16] Cockburn, B., Karniadakis, G., and Shu, C. The development of discontinuous Galerkin methods. In *Discontinuous Galerkin Methods: Theory, Computation and Applications*, vol. 11 of *Lecture Notes in Computational Science and Engineering*. Springer, Berlin, 2000, pp. 1–14.

[17] Cockburn, B., and Shu, C. The local discontinuous Galerkin finite element method for convection diffusion systems. *SIAM Journal on Numerical Analysis 35* (1998), 2440–2463.

[18] Das, P. K., and Deka, G. C. History and evolution of GPU architecture. *Emerging Research Surrounding Power Consumption and Performance Issues in Utility Computing* (2016), 109–135.

[19] Demkowicz, L. *Computing With Hp-adaptive Finite Elements, Vol. 1: One And Two Dimensional Elliptic And Maxwell Problems*. Taylor & Francis Group, 2006.

[20] Demkowicz, L., Kurtz, J., Pardo, D., Paszyński, M., Rachowicz, W., and Zdunek, A. *Computing with Hp-Adaptive Finite Elements, Vol. 2: Frontiers Three Dimensional Elliptic and Maxwell Problems with Applications*. Chapman & Hall/CRC, 2007.

[21] Ding, C. *CUDA Tutorial*. Colorado School of Mines.

[22] Dongarra, J. *LINPACK*. http://www.netlib.org/linpack/.

[23] Duff, I. S., and Reid, J. K. The multifrontal solution of indefinite sparse symmetric linear. *ACM Transactions on Mathematical Software 9*, 3 (1983), 302–325.

[24] Dziekoński, A., Sypek, P., Lamecki, A., and Mrozowski, M. Accuracy, memory, and speed strategies in GPU-based finite-element matrix-generation. *Antennas and Wireless Propagation Letters, IEEE 11* (2012), 1346 – 1349.

[25] Dziekoński, A., Sypek, P., Lamecki, A., and Mrozowski, M. Finite element matrix generation on a GPU. *Progress In Electromagnetics Research 128* (2012), 249–265.

[26] Dziekoński, A., Sypek, P., Lamecki, A., and Mrozowski, M. Generation of large finite-element matrices on multiple graphics processors. *International Journal for Numerical Methods in Engineering 94*, 2 (2013), 204–220.

[27] Falch, T. L., and Elster, A. C. Machine learning based auto-tuning for enhanced opencl performance portability. *CoRR abs/1506.00842* (2015).

[28] Fang, J., Varbanescu, A. L., Sips, H., Zhang, L., Che, Y., and Xu, C. Benchmarking Intel Xeon Phi to guide kernel design. Tech. rep., 2013.

[29] Filipovic, J., Fousek, J., Lakomy, B., and Madzin, M. Automatically optimized GPU acceleration of element subroutines in Finite Element Method. In *Application Accelerators in High Performance Computing (SAAHPC), 2012 Symposium on* (July 2012), pp. 141–144.

[30] Geveler, M., Ribbrock, D., Göddeke, D., Zajac, P., and Turek, S. Towards a complete FEM-based simulation toolkit on GPUs: Unstructured grid finite element geometric multigrid solvers with strong smoothers based on sparse approximate inverses. *Computers & Fluids 80* (2013), 327 – 332. Selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011.

[31] Göddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P., Buijssen, S. H., Grajewski, M., and Turek, S. Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. *Parallel Computing 33*, 10-11 (November 2007), 685–699.

[32] Göddeke, D., Strzodka, R., and Turek, S. Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations. *International Journal of Parallel, Emergent and Distributed Systems 22*, 4 (2007), 221–256.

[33] Göddeke, D., Wobker, H., Strzodka, R., Mohd-Yusof, J., McCormick, P., and Turek, S. a. Co–processor acceleration of an unmodified parallel solid mechanics code with FEASTGPU. *International Journal of Computational Science and Engineering 4*, 4 (Oct. 2009), 254–269.

[34] Goodwins, R. Intel unveils many-core Knights platform for HPC. *ZdNet* (2010). Accessed on 27th November 2015.

[35] Govindaraju, N. K., Larsen, S., Gray, J., and Manocha, D. A memory model for scientific algorithms on graphics processors. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 2006), SC '06, ACM.

[36] Graczyk, R. Intel Iris Pro 5200 test; Crysis 3 na integrze? Accessed on 10th Febuary 2015.

[37] Gruber, R., and Keller, V. *HPC@GreenIT*. Springer Berlin Heidelberg, 2010.

[38] Halfacree, G. AMD announces heterogeneous queuing tech. *http://www.bit-tech.net/* (22nd October 2013). Accessed on 10th Febuary 2015.

[39] Hanzlikova, N., and Rodrigues, E. R. A novel finite element method assembler for co-processors and accelerators. In *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms* (New York, NY, USA, 2013), ACM, pp. 1:1–1:8.

[40] Heirich, A., and Bavoil, L. Deferred pixel shading on the Playstation 3. Accessed on 10th Febuary 2015.

[41] Howes, L.; Munshi, A. *The OpenCL Specification.* Khronos OpenCL Working Group, 2014. version 2.0, revision 26.

[42] HSA Foundation. *http://www.hsafoundation.com*, 2013. Accessed on 10th Febuary 2015.

[43] IBM. *Cell Broadband Engine Programming Handbook Including the PowerXCell 8i Processor*, 2008.

[44] Intel Corporation. *https://ark.intel.com/pl.* Specyfikacje produktów Intel.

[45] Intel Corporation. *Writing Optimal OpenCL Code with Intel OpenCL SDK*, rev. 1.3 ed., 2010-2011.

[46] Intel Corporation. *A Guide to Vectorization with Intel C++ Compilers*, 2012.

[47] Intel Corporation. *OpenCL 2.0 Shared Virtual Memory Overview*, September 2014.

[48] Intel Corporation. *OpenCL Design and Programming Guide for the Intel Xeon Phi Coprocessor*, January 2014. Accessed on 28th April 2014.

[49] Intel Corporation. *Intel C++ Compiler 16.0 User and Reference Guide*, 2015.

[50] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, January 2016.

[51] Irons, B. M. A frontal solution scheme for finite element analysis. *International Journal for Numerical Methods in Engineering 2*, 4-5 (1970), 5–32.

[52] Ismail, L., and Guerchi, D. Performance evaluation of convolution on the cell broadband engine processor. *IEEE Transactions on Parallel and Distributed Systems 22*, 2 (Feb 2011), 337–351.

[53] Jeffers, J., and Reinders, J. *Intel Xeon Phi Coprocessor High Performance Programming*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013.

[54] Jeffers, J., and Reinders, J. *Intel Xeon Phi Coprocessor High Performance Programming, 1st Edition.* Morgan Kaufmann, 2013.

[55] Johan, Z., and Hughes, T. A globally convergent matrix-free algorithm for implicit time-marching schemes arising in finite element analysis in fluids. *Computer Methods in Applied Mechanics and Engineering 87* (1991), 281–304.

[56] Johnson, C. *Numerical Solution of Partial Differential Equations by the Finite Element Method*. Cambridge University Press, 1987.

[57] Kanter, D. Intels Haswell CPU microarchitecture. *Real World Technologies* (November 2012).

[58] Kirby, R. C., Knepley, M. G., Logg, A., and Scott, L. R. Optimizing the evaluation of finite element matrices. *SIAM J. Scientific Computing 27*, 3 (2005), 741–758.

[59] Kirby, R. C., and Logg, A. A compiler for variational forms. *ACM Transactions on Mathematical Software 32*, 3 (2006).

[60] Kitowski, J., Rozen, T., Boryczko, K., and Alda, W. Particle simulations of two-phase flows on cell broadband engine. In *2008 International Symposium on Parallel and Distributed Computing* (July 2008), pp. 439–443.

[61] Klöckner, A., Warburton, T., Bridge, J., and Hesthaven, J. S. Nodal discontinuous galerkin methods on graphics processors. *J. Comput. Phys. 228* (November 2009), 7863–7882.

[62] Knepley, M. G., and Terrel, A. R. Finite element integration on GPUs. *ACM Trans. Math. Softw. 39*, 2 (Feb. 2013), 10:1–10:13.

[63] Komatitsch, D., Erlebacher, G., Göddeke, D., and Michéa, D. High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. *Journal of Computational Physics 229*, 20 (2010), 7692 – 7714.

[64] Komatitsch, D., Michéa, D., and Erlebacher, G. Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. *Journal of Parallel and Distributed Computing 69*, 5 (2009), 451–460.

[65] Krużel, F., and Banaś, K. Finite element numerical integration on Xeon Phi coprocessor. In *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems* (2014), M. P. M. Ganzha, L. Maciaszek, Ed., vol. 2 of *Annals of Computer Science and Information Systems*, IEEE, pp. pages 603–612.

[66] Krużel, F., and Banaś, K. Finite element numerical integration on PowerXCell processors. In *PPAM'09: Proceedings of the 8th international conference on Parallel processing and applied mathematics* (Berlin, Heidelberg, 2010), Springer-Verlag, pp. 517–524.

[67] Krużel, F., and Banaś, K. Vectorized OpenCL implementation of numerical integration for higher order finite elements. *Computers and Mathematics with Applications 66*, 10 (2013), 2030–2044.

[68] Krużel, F., and Banaś, K. AMD APU systems as a platform for scientific computing. *Computer Methods in Materials Science 15*, 2 (2015), 362–369.

[69] Krużel, F., Madej, Ł., Perzyński, K., and Banaś, K. Development of three-dimensional adaptive mesh generation for multiscale applications. *International Journal for Multiscale Computational Engineering 12*, 3 (2014), 257–269.

[70] Kumar, S. Fundamental limits to Moore's Law. *arXiv:1511.05956* (2015).

[71] Kushida, N. Element-wise implementation of iterative solvers for FEM problems on the Cell processor. In *Proceedings of the 19th International Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP 2011, Ayia Napa, Cyprus, 9-11 February 2011* (2011), Y. Cotronis, M. Danelutto, and G. A. Papadopoulos, Eds., IEEE Computer Society, pp. 401–408.

[72] Kyriazis, G. Heterogeneous system architecture: A technical review. Tech. rep., AMD, 2012. revision 1.0.

[73] Landaverde, R., Zhang, T., Coskun, A., and Herbordt, M. An investigation of unified memory access performance in CUDA. *IEEE High Performance Extreme Computing* (2014).

[74] Logg, A., Mardal, K.-A., Wells, G. N., et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[75] Logg, A., and Wells, G. N. Dolfin: Automated finite element computing. *ACM Trans. Math. Softw. 37*, 2 (Apr. 2010), 20:1–20:28.

[76] Luporini, F., Varbanescu, A. L., Rathgeber, F., Bercea, G., Ramanujam, J., Ham, D. A., and Kelly, P. H. J. COFFEE: an optimizing compiler for finite element local assembly. *CoRR abs/1407.0904* (2014).

[77] Luporini, F., Varbanescu, A. L., Rathgeber, F., Bercea, G.-T., Ramanujam, J., Ham, D. A., and Kelly, P. H. J. Cross-loop optimization of arithmetic intensity for finite element local assembly. *ACM Trans. Archit. Code Optim. 11*, 4 (Jan. 2015), 57:1–57:25.

[78] Mafi, R., and Sirouspour, S. GPU-based acceleration of computations in nonlinear finite element deformation analysis. *International Journal for Numerical Methods in Biomedical Engineering 30*, 3 (2014), 365–381.

[79] Mamza, J., Makyla, P., Dziekoński, A., Lamecki, A., and Mrozowski, M. Multi-core and multiprocessor implementation of numerical integration in Finite Element Method. In *Microwave Radar and Wireless Communications (MIKON), 2012 19th International Conference on* (2012), vol. 2, pp. 457 – 461.

[80] Markall, G. R., Ham, D. A., and Kelly, P. H. Towards generating optimised finite element solvers for GPUs from high-level specifications. *Procedia Computer Science 1*, 1 (2010), 1815 – 1823. ICCS 2010.

[81] Markall, G. R., Rathgeber, F., Mitchell, L., Loriant, N., Bertolli, C., Ham, D. A., and Kelly, P. H. J. *Supercomputing: 28th International Supercomputing Conference, ISC 2013, Leipzig, Germany, June 16-20, 2013. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, ch. Performance-Portable Finite Element Assembly Using PyOP2 and FEniCS, pp. 279–289.

[82] Markall, G. R., Slemmer, A., Ham, D. A., Kelly, P. H. J., Cantwell, C. D., and Sherwin, S. J. Finite element assembly strategies on multi-core and many-core architectures. *International Journal for Numerical Methods in Fluids 71*, 1 (2013), 80–97.

[83] McCalpin, J. Memory bandwidth and machine balance in current high performance computers. 19–25.

[84] McCalpin, J. How to measure flops using vtunes? Intel Devoloper Zone Forum, 2014.

[85] McCalpin, J. D. Memory bandwidth and machine balance in current high performance computers. *IEEE Technical Committee on ComputerArchitecture (TCCA) Newsletter* (December 1995), 19–25.

[86] Melenk, J., Gerdes, K., and Schwab, C. Fully discrete *hp*-finite elements I: fast quadrature. *Computer Methods in Applied Mechanics and Engineering 190* (2001), 4339–64.

[87] Michalik, K., Banaś, K., Płaszewski, P., and Cybułka, P. ModFEM – a computational framework for parallel adaptive finite element simulations. *Computer Methods in Materials Science 13*, 1 (2013), 3–8.

[88] Michalik, K., Banaś, K., Płaszewski, P., and Cybułka, P. Modular FEM framework ModFEM for generic scientific parallel simulations. *Computer Science 14*, 3 (2013).

[89] Moore, G. E. Cramming more components onto integrated circuits. *Electronics 38*, 8 (19 April 1965).

[90] no X. Několik novinek o kaveri: GDDR5, snížené TDP a možná i TrueAudio. *Deep in IT* (2013).

[91] Nvidia Corporation. *NVIDIAs Next Generation CUDA Compute Architecture: Kepler GK110*, 2012. Whitepaper.

[92] Nvidia Corporation. *CUDA C Programming Guide Design Guide*, August 2014. version 6.5.

[93] Nvidia Corporation. NVIDIA NVLink High-Speed Interconnect: Application Performance. Whitepaper, November 2014.

[94] Nvidia Corporation. *NVIDIA CUDA C Programming Guide*, 2015. Version 7.5.

[95] Nvidia Corporation. *PROFILER USER'S GUIDE*, September 2015.

[96] Olas, T., Mleczko, W. K., Nowicki, R. K., Wyrzykowski, R., and Krzyzak, A. *Adaptation of RBM Learning for Intel MIC Architecture*. Springer International Publishing, Cham, 2015, pp. 90–101.

[97] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*, version 4.5 ed., November 2015.

[98] Papermaster, M. The surround computing era. Tech. rep., 28 August 2012.

[99] Paszyński, M., Pardo, D., Paszyńska, A., and Demkowicz, L. F. Out-of-core multi-frontal solver for multi-physics hp adaptive problems. *Procedia CS 4* (2011), 1788–1797.

[100] Płaszewski, P., and Banaś, K. Performance analysis of iterative solvers of linear equations for hp-adaptive finite element method. In *Proceedings of the International Conference on Computational Science, ICCS 2013, Barcelona, Spain, 5-7 June,*

2013 (2013), V. N. Alexandrov, M. Lees, V. V. Krzhizhanovskaya, J. Dongarra, and P. M. A. Sloot, Eds., vol. 18 of *Procedia Computer Science*, Elsevier, pp. 1584–1593.

[101] Rathgeber, F., Ham, D. A., Mitchell, L., Lange, M., Luporini, F., McRae, A. T. T., Bercea, G., Markall, G. R., and Kelly, P. H. J. Firedrake: automating the Finite Element Method by composing abstractions. *CoRR abs/1501.01809* (2015).

[102] Rathgeber, F., Markall, G. R., Mitchell, L., Loriant, N., Ham, D. A., Bertolli, C., and Kelly, P. H. J. Pyop2: A high-level framework for performance-portable simulations on unstructured meshes. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:* (Nov 2012), pp. 1116–1123.

[103] Reguly, I., and Giles, M. Efficient sparse matrix-vector multiplication on cache-based GPUs. In *Innovative Parallel Computing (InPar), 2012* (May 2012), pp. 1–12.

[104] Reguly, I., and Giles, M. Finite element algorithms and data structures on graphical processing units. *International Journal of Parallel Programming* (2013), 1–37.

[105] Rojek, K., and Szustak, Ł. Adaptation of double-precision matrix multiplication to the Cell Broadband Engine architecture. In *PPAM'09: Proceedings of the 8th international conference on Parallel processing and applied mathematics* (Berlin, Heidelberg, 2010), Springer-Verlag, pp. 535–546.

[106] Roth, F. *System Administration for the Intel Xeon Phi Coprocessor*. Intel Corporation, 2013.

[107] Rul, S., Vandierendonck, H., D'Haene, J., and De Bosschere, K. An experimental study on performance portability of OpenCL kernels. In *Application Accelerators in High Performance Computing, 2010 Symposium, Papers* (Knoxville, TN, USA, 2010), p. 3.

[108] Rumpf, M., and Strzodka, R. Graphics processor units: New prospects for parallel computing. In *Numerical Solution of Partial Differential Equations on Parallel Computers*, A.M.Bruaset and A.Tveito, Eds., vol. 51 of *Lecture Notes in Computational Science and Engineering*. Springer, 2005, pp. 89–134.

[109] Saule, E., Kaya, K., and Çatalyürek, Ü. V. Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi. *CoRR abs/1302.1078* (2013).

[110] Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., and Dubey, P. Larrabee: a many-core x86 architecture for visual computing. *SIGGRAPH 08: ACM SIGGRAPH 2008 papers* (2008), 1–15.

[111] Sherwin, S., and Karniadakis, G. Tetrahedral *hp* finite elements: algorithms and flow simulations. *Journal of Computational Physics 124* (1996), 14–45.

[112] Smith, R. Amd radeon hd 7970 review: 28nm and graphics core next, together as one. *AnandTech* (December 2011).

[113] Strassburg, J. Intel Xeon and Intel Xeon Phi processor update, May 2016. Intel HPC Software Workshop Series 2016.

[114] Szustak, Ł., Rojek, K., and Gepner, P. *Using Intel Xeon Phi Coprocessor to Accelerate Computations in MPDATA Algorithm.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 582–592.

[115] Szustak, Ł., Rojek, K., Olas, T., Kuczynski, L., Halbiniak, K., , and Gepner, P. Adaptation of MPDATA heterogeneous stencil computation to Intel Xeon Phi coprocessor,. *Scientific Programming vol. 2015* (2015), 14.

[116] Tendler, J. M., Dodson, J. S., Fields, J. S., Le, H., and Sinharoy, B. Power4 system microarchitecture. *IBM Journal of Research and Development 46*, 1 (Jan 2002), 5–25.

[117] Ungerer, T., Robic, B., and Silc, J. Multithreaded processors. *The Computer Journal 45*, 3 (2002), 320–348.

[118] Van Winkle, W. AMD Fusion: How It Started, Where Its Going, And What It Means. *Tom's HARDWARE* (2012). Accessed on 10th Febuary 2015.

[119] Volkov, V., and Demmel, J. W. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (Piscataway, NJ, USA, 2008), SC '08, IEEE Press, pp. 31:1–31:11.

[120] Vos, P. E. J., Sherwin, S. J., and Kirby, R. M. From h to p efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low- and high-order discretisations. *J. Comput. Phys. 229* (July 2010), 5161–5181.

[121] Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., and Demmel, J. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Comput. 35*, 3 (2009), 178–194.

[122] Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., and Yelick, K. Scientific computing kernels on the Cell processor. *Int. J. Parallel Program. 35*, 3 (2007), 263–298.

[123] Woźniak, M., Łoś, M., Paszyński, M., and Demkowicz, L. Fast parallel integration for three dimensional discontinuous petrov galerkin method. *Procedia Computer Science 101* (2016), 8 – 17. 5th International Young Scientist Conference on Computational Science, YSC 2016, 26-28 October 2016, Krakow, Poland.

[124] Wu, W., and Heng, P. A. A hybrid condensed finite element model with GPU acceleration for interactive 3d soft tissue cutting. *Computer Animation and Virtual Worlds 15*, 3-4 (2004), 219–227.

[125] Wyrzykowski, R., Rojek, K., and Szustak, Ł. Model-driven adaptation of double-precision matrix multiplication to the Cell processor architecture. *Parallel Computing 38*, 4-5 (2012), 260–276.

[126] Zienkiewicz, O., and Taylor, R. *Finite element method. Vol 1-3.* Butterworth Heinemann, London, 2000.

[127] Zienkiewicz, O. C., and Zhu, J. Z. The superconvergent patch recovery (SPR) and adaptive finite element refinement. *Comput. Methods Appl. Mech. Eng. 101*, 1-3 (Dec. 1992), 207–224.

Spis rysunków

Spis tabel

Dodatek

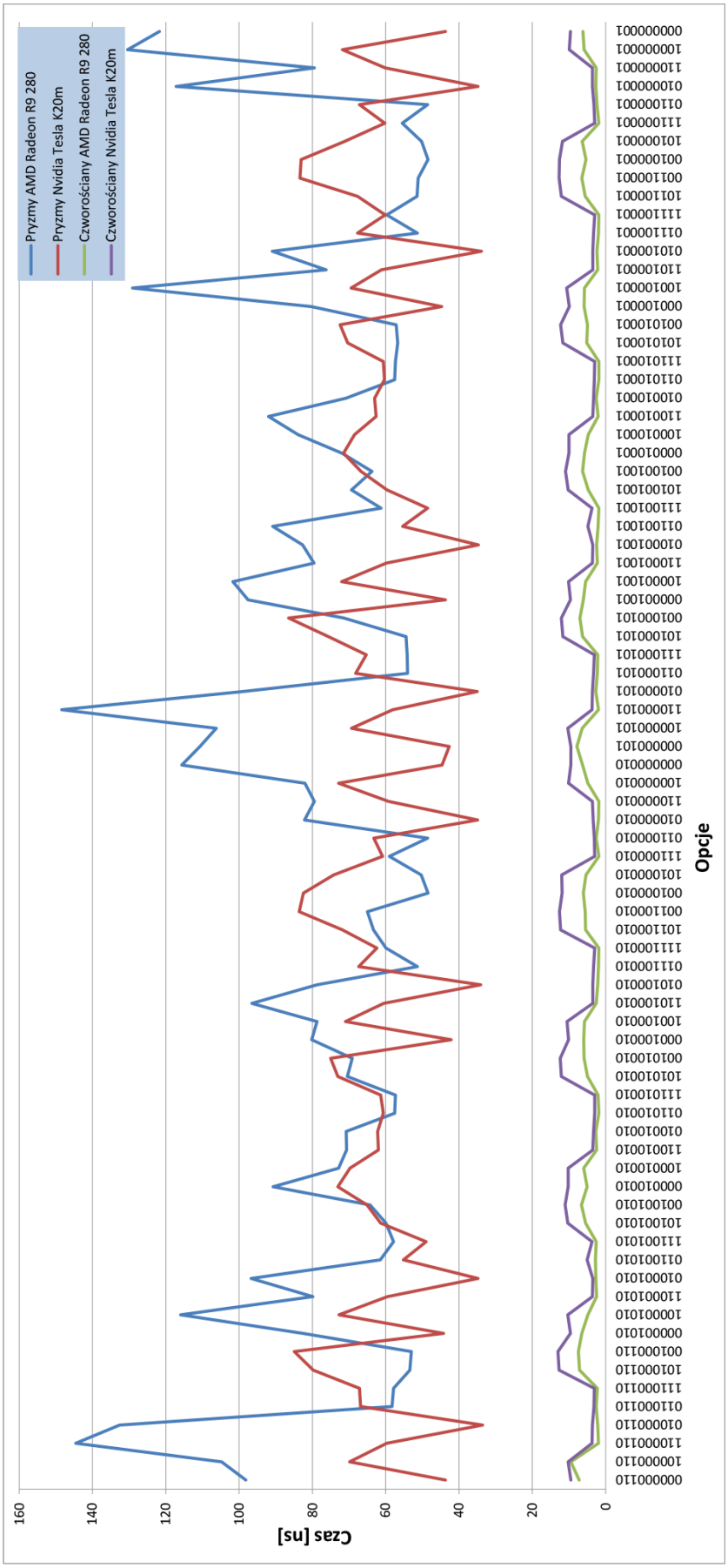
Dodatek A

Wyniki automatycznego tuningu dla procesorów graficznych

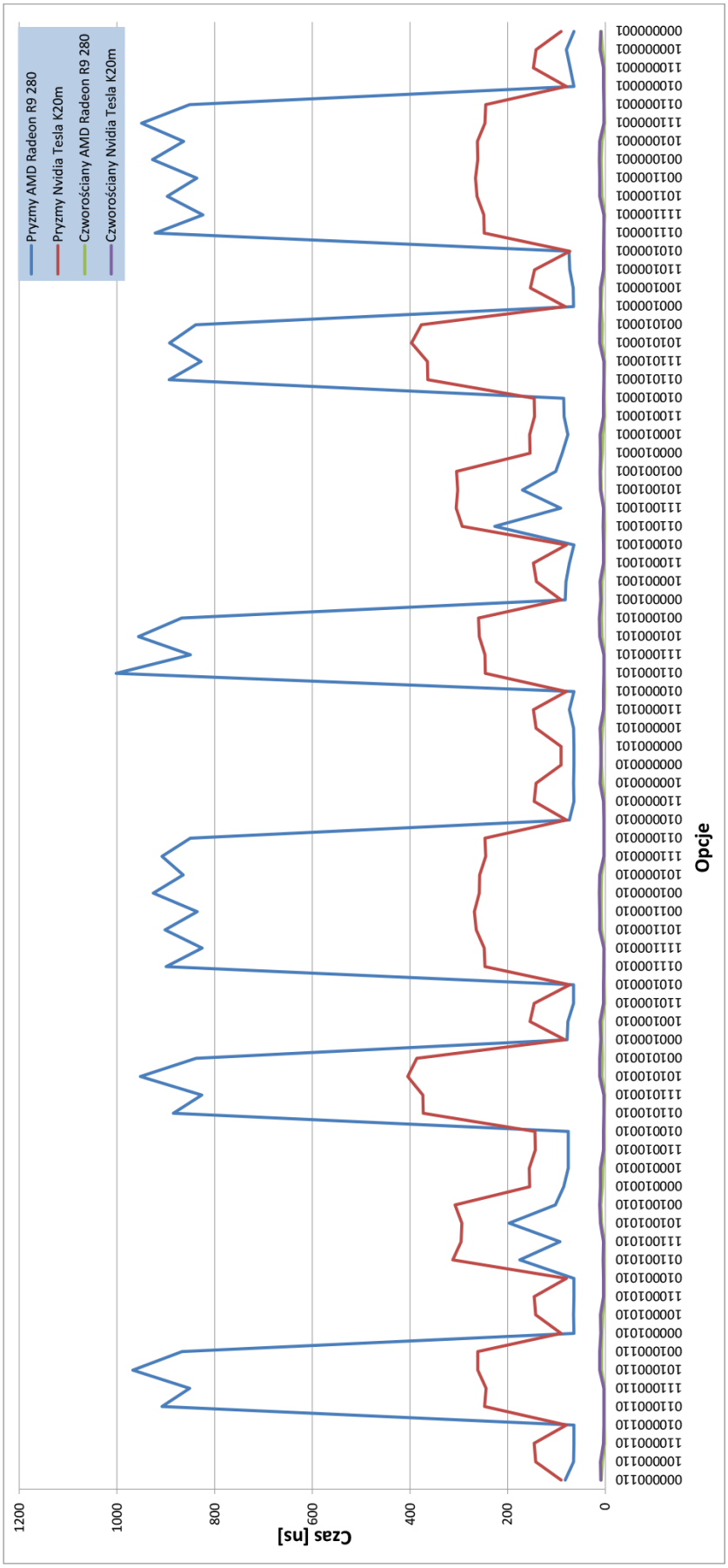
A.1. Problem Poissona

Wyniki dla algorytmu SQS i zadania Poissona zebrano na rysunku A.1. Tak jak w przypadku algorytmu QSS możemy zauważyć powtarzalność wyników dla opcji dopasowania tablic. Uwagę zwraca także stabilność wyniku dla elementów czworościennych, gdzie wyłączenie opcji zapisu ciągłego powoduje regularny wzrost czasu wykonania dla obu badanych architektur. Dla elementów pryzmatycznych oraz karty Radeon, uzyskano ponownie najlepszy rezultat dla opcji z wyłączeniem ciągłego odczytu oraz włączeniem CASFD. Co ciekawe opcja ciągłego zapisu nie odgrywała tu również dużej roli. Dla architektury Nvidia, najlepszy czas uzyskała opcja z przechowywaniem macierzy wynikowych w pamięci wspólnej - w przeciwieństwie do opcji QSS, tablice takie w tym wariancie algorytmu, służą jedynie do przechowywania wierszy macierzy wynikowych więc zajmują dużo mniej miejsca. Dla takiej kombinacji algorytmu SQS, opcja ciągłego odczytu nie musi być włączona.

Jest to zrozumiałe, gdyż przy opcji ciągłego odczytu wykorzystywana jest tymczasowa tablica w pamięci współdzielonej, która w tym przypadku nie jest do niczego innego potrzebna i generuje jedynie dodatkowe odczyty, które w przypadku wyłączenia tej opcji, są zastąpione przez odpowiednie użycie pamięci cache. Dla akceleratora AMD i elementów pryzmatycznych, najgorsze wyniki uzyskano przy używaniu pamięci wspólnej do przechowywania wynikowych macierzy (STIFF) oraz przy wyłączeniu opcji CASFD. W przeciwieństwie do Radeona, włączenie obliczania pochodnych funkcji kształtu dla Tesli, powoduje znaczący spadek wydajności. W połączeniu z brakiem ciągłego zapisu, dało to najgorszy wynik dla tej karty. Dla elementów czworościennych najlepszy wynik, dla obu badanych akceleratorów, uzyskano dla wariantów z ciągłym odczytem i zapisem, włączonym CASFD oraz użyciem pamięci wspólnej dla danych problemowych lub geometrycznych. Tak jak i w poprzednim przypadku, karta Radeon okazuje się być szybsza dla elementów czworościennych



Rysunek A.1: Automatyczny tuning parametryczny dla zadania Poissona oraz algorytmu SQS



Rysunek A.2: Automatyczny tuning parametryczny dla zadania Poissona oraz algorytmu SSQ

(średnio o 75%), a karta Tesla jest szybsza dla pryzm (średnio o 25%). Algorytm SQS okazuje się być nieznacznie szybszy od QSS dla elementów czworościennych (ok 2% dla AMD oraz 13% dla Nvidii). Za to dla zagadnień z nieliniową geometrią, obserwujemy znaczący spadek wydajności (o ok 70% w obu przypadkach).

Dla algorytmu SSQ, najgorsze spadki wydajności związane są z włączeniem opcji CASFD dla elementów pryzmatycznych w związku z redundantnym powtarzaniem obliczeń dla każdego punktu Gaussa (Rys. A.2). Taka sytuacja nie występuje w przypadku elementów czworościennych gdzie obliczenia te odbywają się poza główną pętlą po funkcjach kształtu. Dla akceleratora Radeon, najlepsze wyniki uzyskano używając pamięci wspólnej dla funkcji kształtu i ich pochodnych dla pryzm oraz dla danych geometrycznych dla czworościanów. W przypadku karty firmy Nvidia, najlepszy wynik dla elementów pryzmatycznych otrzymano przy wyłączeniu odczytu ciągłego oraz użyciu pamięci wspólnej na współczynniki problemowe. Dla czworościanów, najlepsze efekty uzyskano używając ciągłego odczytu i zapisu przy równoczesnym nie używaniu pamięci współdzielonej. Tak jak w poprzednim przypadku akcelerator Nvidia okazał się być szybszy dla elementów geometrycznie nieliniowych (prawie dwukrotnie), a wolniejszy (ok. 50%) dla elementów czworościennych. W porównaniu z poprzednimi opcjami, algorytm SSQ jest wolniejszy od SQS dla pryzm (25% dla Radeona oraz ponad 50% dla Tesli), a dla czworościanów daje praktycznie te same wyniki.

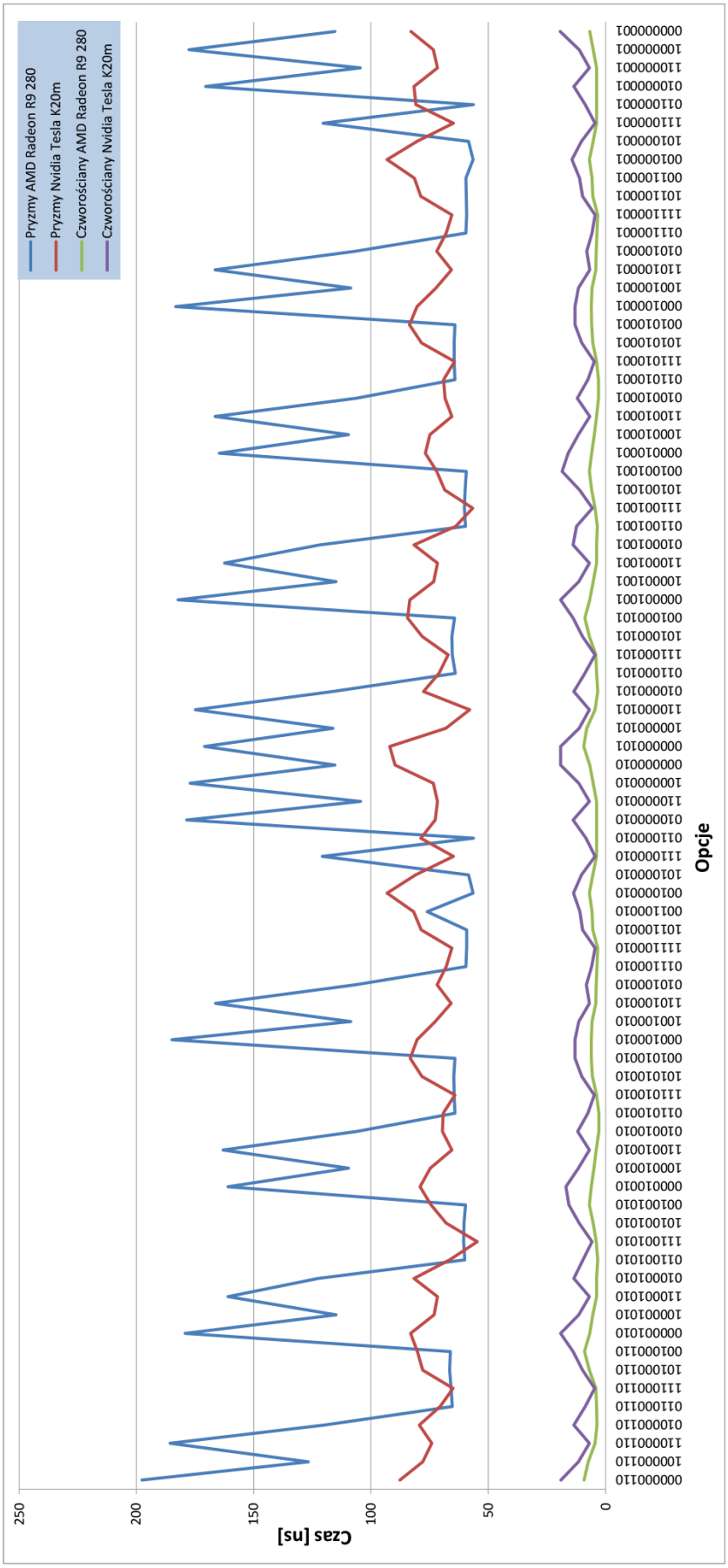
A.2. Problem konwekcji-dyfuzji

Wyniki działania algorytmu SQS prezentuje rysunek A.3. Tak samo jak w przypadku zadania Poissona, uwagę zwraca stabilność wyników dla elementów czworościennych, niezależnie od opcji - szczególnie dla karty Radeon. Dla elementów pryzmatycznych i tej karty, widoczne są duże spadki wydajności związane z wyłączaniem opcji CASFD. Charakterystyczny kształt "piły" związany jest z naprzemiennym włączaniem i wyłączeniem opcji ciągłego odczytu i zapisu - okazuje się, że najlepszą kombinacją jest włączanie jedynie jednej z tych dwóch opcji. Najlepszy rezultat uzyskano dla włączonego zapisu ciągłego oraz opcji CASFD przy nie używaniu pamięci wspólnej. Dla akceleratora Nvidia, wyniki są stabilniejsze, a najlepszy rezultat uzyskano dla ciągłego zapisu i odczytu, włączonej opcji CASFD oraz przechowywaniem pochodnych funkcji kształtu w pamięci wspólnej. W obu przypadkach, najgorsze rezultaty uzyskano nie używając w ogóle ciągłego sposobu komunikacji z pamięcią. W przypadku Radeona dodatkowo przechowywanie finalnych macierzy

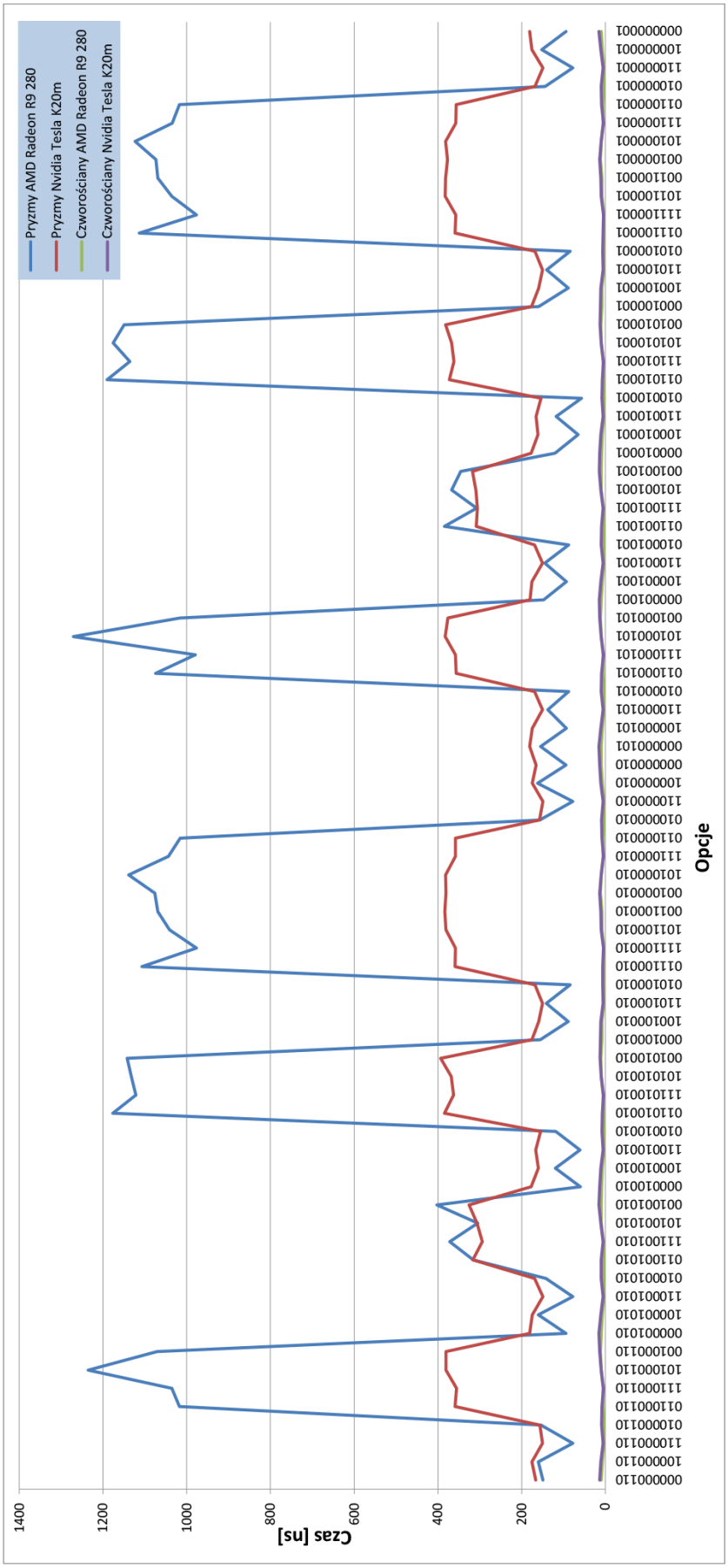
w pamięci wspólnej, oraz wyłączenie opcji CASFD, spowodowało spadek wydajności, a w przypadku Tesli taki sam efekt przyniosło samo włącznie opcji CASFD.

W przypadku elementów czworościennych na karcie Tesla uzyskano taki sam najlepszy rezultat jak w przypadku algorytmu QSS, czyli dla włączonych ciągłych odczytów i zapisów, opcji CASFD oraz PDE. Dla akceleratora AMD najlepsze rezultaty uzyskano wyłączając ciągły odczyt, oraz używając pamięci wspólnej dla danych geometrycznych - w tym przypadku dodatkowo opcja CASFD nie ma wpływu na wykonanie. Dla obu kart uzyskano najgorsze rezultaty dla wyłączonych wszystkich opcji oprócz przechowywania finalnej macierzy (STIFF) w pamięci wspólnej. Dla algorytmu SQS wyniki dla pryzm zajmują tyle samo czasu dla obu akceleratorów i są 60% wolniejsze dla Radeona i 73% wolniejsze dla Tesli. W przypadku elementów czworościennych dla Radeona wyniki są gorsze o 16%, a dla Tesli o 13% lepsze.

Algorytm SSQ, tak jak w przypadku zadania Poissona, charakteryzują duże spadki wydajności związane z włączeniem opcji CASFD dla elementów pryzmatycznych (Rys. A.4). Dla karty Radeon uzyskano najlepsze rezultaty dla wyłączonego ciągłego odczytu, włączonego zapisu (CW) oraz dla użycia pamięci wspólnej dla danych geometrycznych, zarówno dla elementów czworościennych jak i pryzmatycznych. Tak samo w przypadku najgorszych wyników, włączenie opcji CASFD wraz z wyłączonym zapisem (CW), daje ten sam efekt dla różnego typu elementów. W przypadku karty Nvidia i elementów pryzmatycznych, najlepsze efekty uzyskano nie używając pamięci wspólnej, przy równoczesnym ciągłym odczycie i zapisie. W przypadku czworościanów, dodatkowo należy włączyć opcję CASFD. Najgorsze wyniki uzyskano w przypadku rezygnacji z ciągłego odczytu i zapisu oraz użycia pamięci wspólnej dla danych geometrycznych (GEO) lub funkcji kształtu (SHP).



Rysunek A.3: Automatyczny tuning parametryczny dla zadania konwekcji-dyfuzji oraz algorytmu SQS

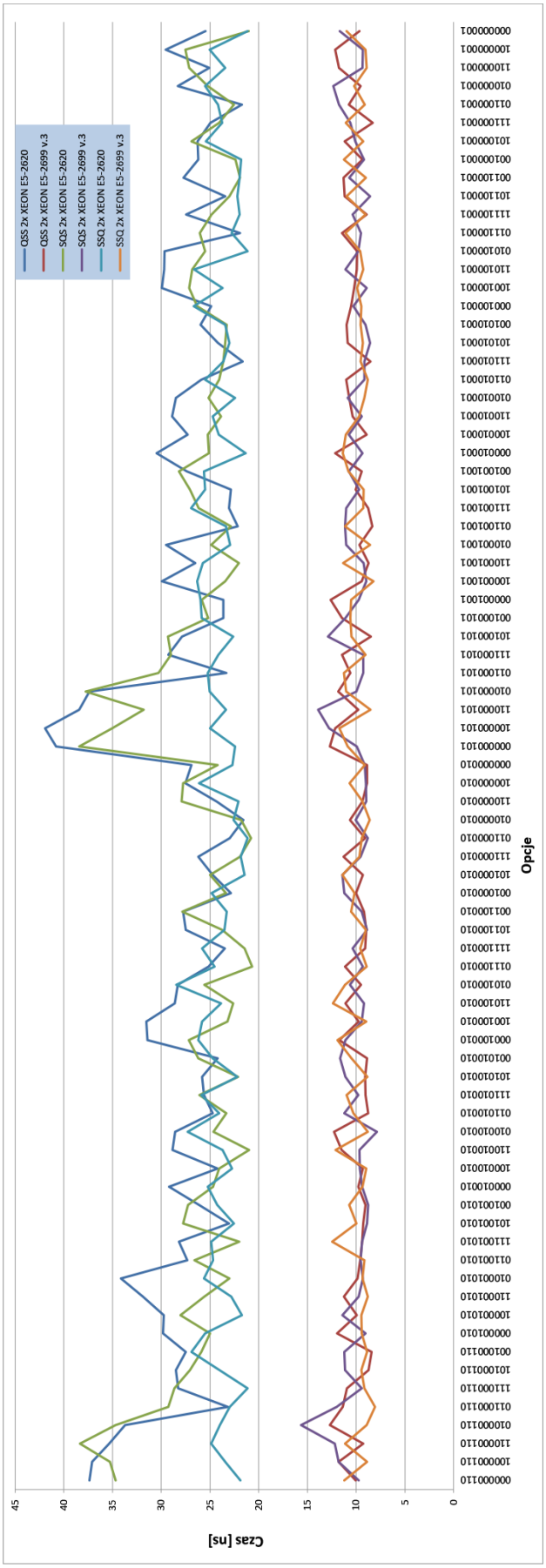


Rysunek A.4: Automatyczny tuning parametryczny dla zadania konwekcji-dyfuzji oraz algorytmu SSQ

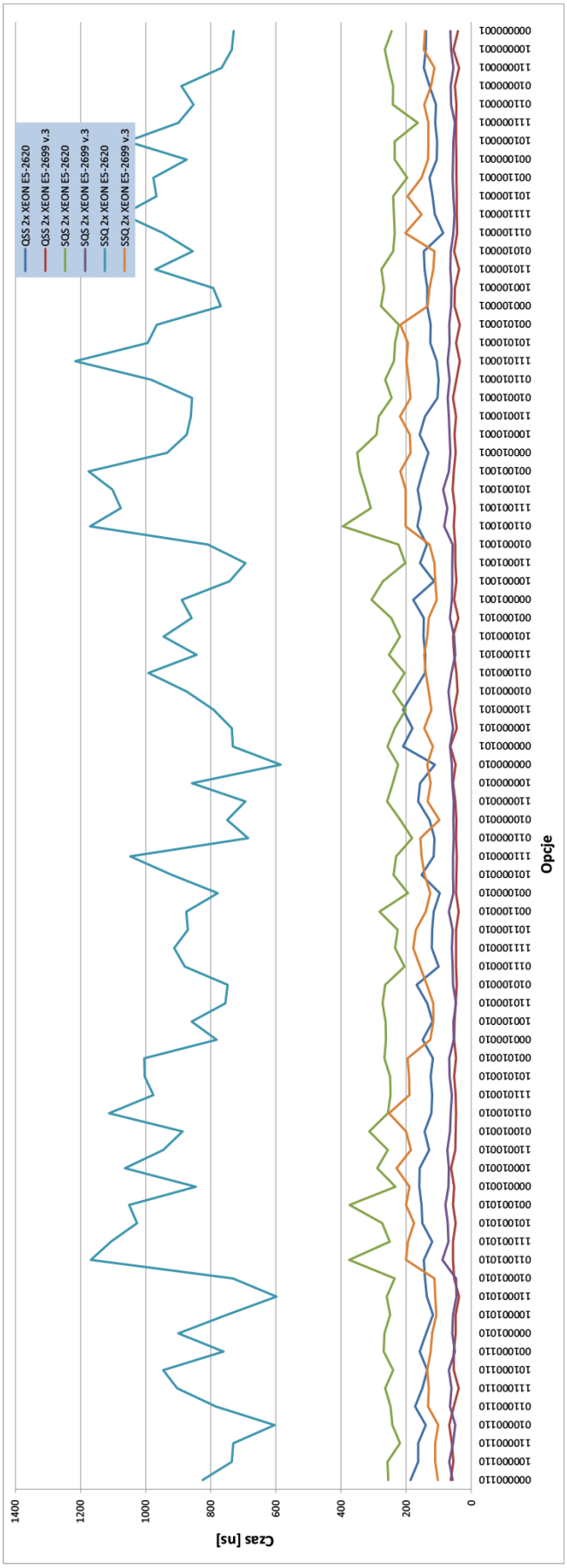
Dodatek B

Wyniki automatycznego tuningu dla procesorów ogólnego przeznaczenia

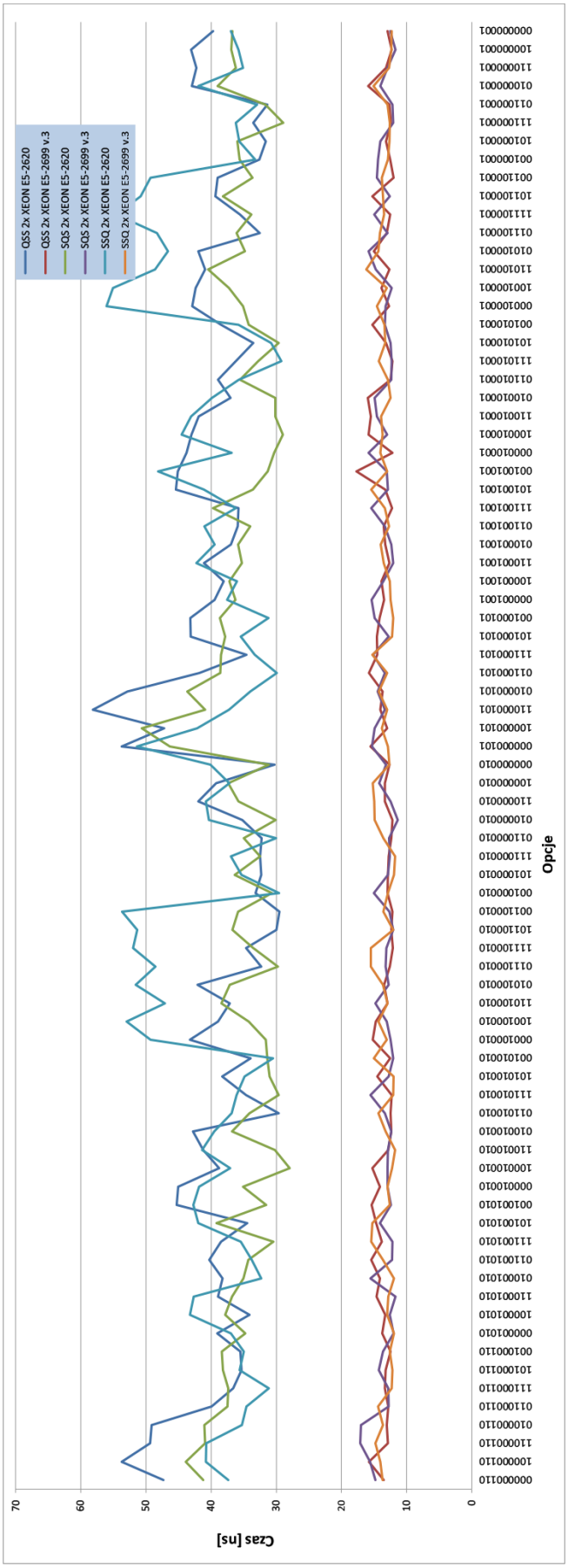
W przypadku automatycznego tuningu na CPU możemy zauważyć grupowanie się wyników względem architektury. Dla procesorów Haswell wyniki poszczególnych wersji są do siebie zbliżone - szczególnie dla elementów czworościennych (Rys. B.1 i B.3). W przypadku procesora Sandy Bridge wyniki dla zadania Poissona i elementów czworościennych zwiększają się nagle przy przechowywaniu macierzy sztywności w pamięci wspólnej w przypadku wersji QSS oraz SQS - jako że w przypadku SSQ, macierz zajmuje tylko jedną zmienną, to ten problem tam nie występuje. W przypadku zadania konwekcji-dyfuzji, większe fluktuacje obserwujemy przy przechowywaniu współczynników problemowych i danych geometrycznych w pamięci wspólnej. Oba te przypadki świadczą o tym, iż jako pamięć wspólna w modelu OpenCL traktowany jest cache CPU, dzięki temu dla procesora o bardzo dużym cache (Haswell) tego typu wahania nie występują. W przypadku zadania konwekcji-dyfuzji i elementów pryzmatycznych, widoczny jest dosyć duży rozrzut wyników dla każdej z architektur, ale wymienione wyżej prawidłowości dotyczące pamięci cache są nadal widoczne.



Rysunek B.1: Automatyczny tuning parametryczny dla zadania Poissona i elementów czworościennych na procesorach CPU



Rysunek B.2: Automatyczny tuning parametryczny dla zadania konwekcji-dyfuzji i elementów pryzmatycznych na procesorach CPU

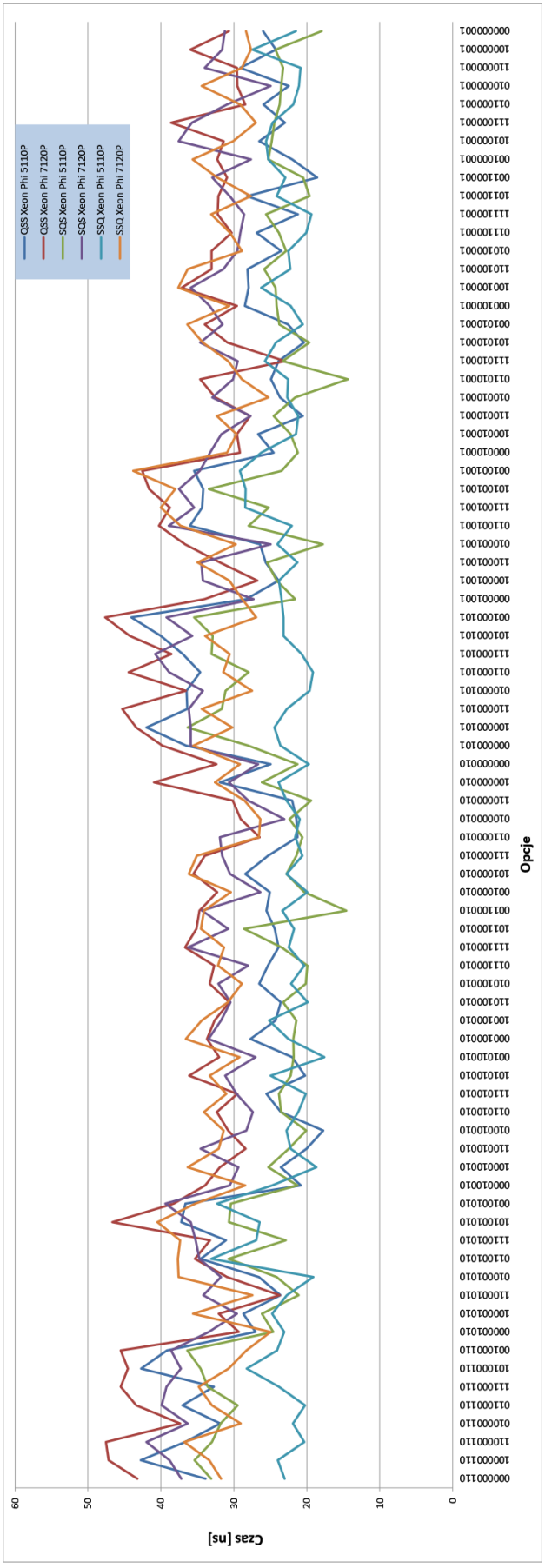


Rysunek B.3: Automatyczny tuning parametryczny dla zadania konwekcji-dyfuzji i elementów czworościennych na procesorach CPU

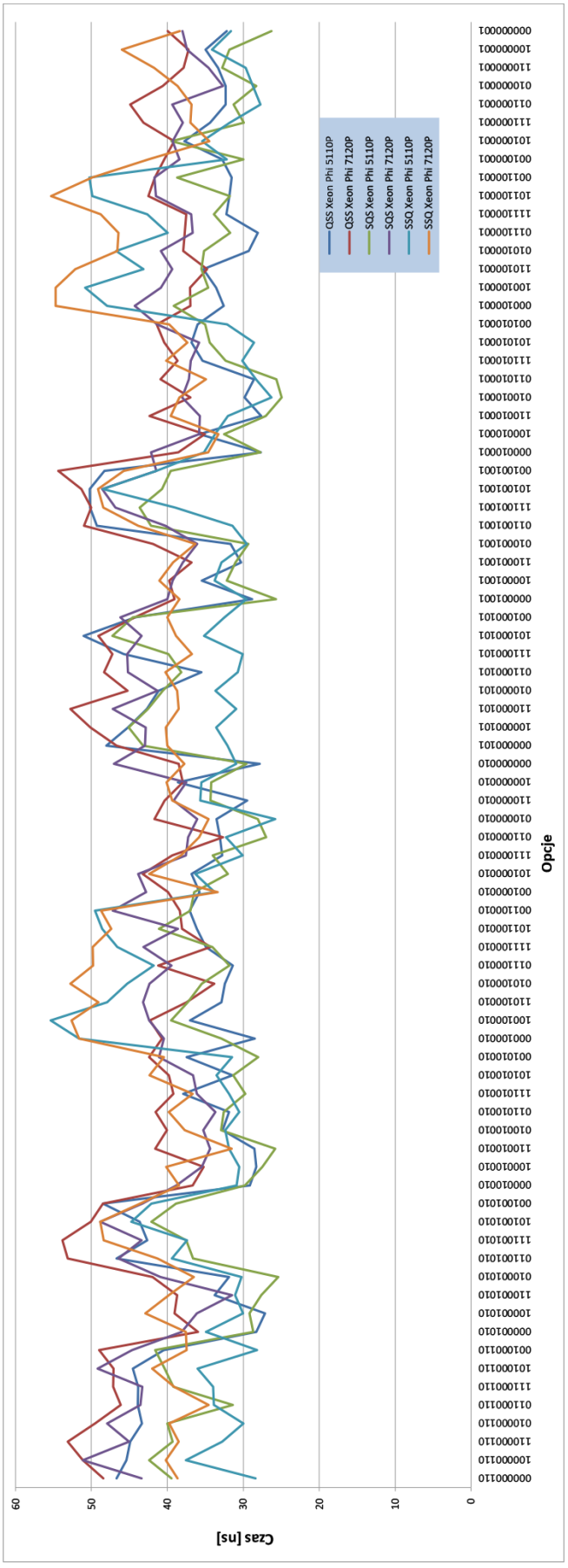
Dodatek C

Wyniki automatycznego tuningu dla koprocesorów Xeon Phi

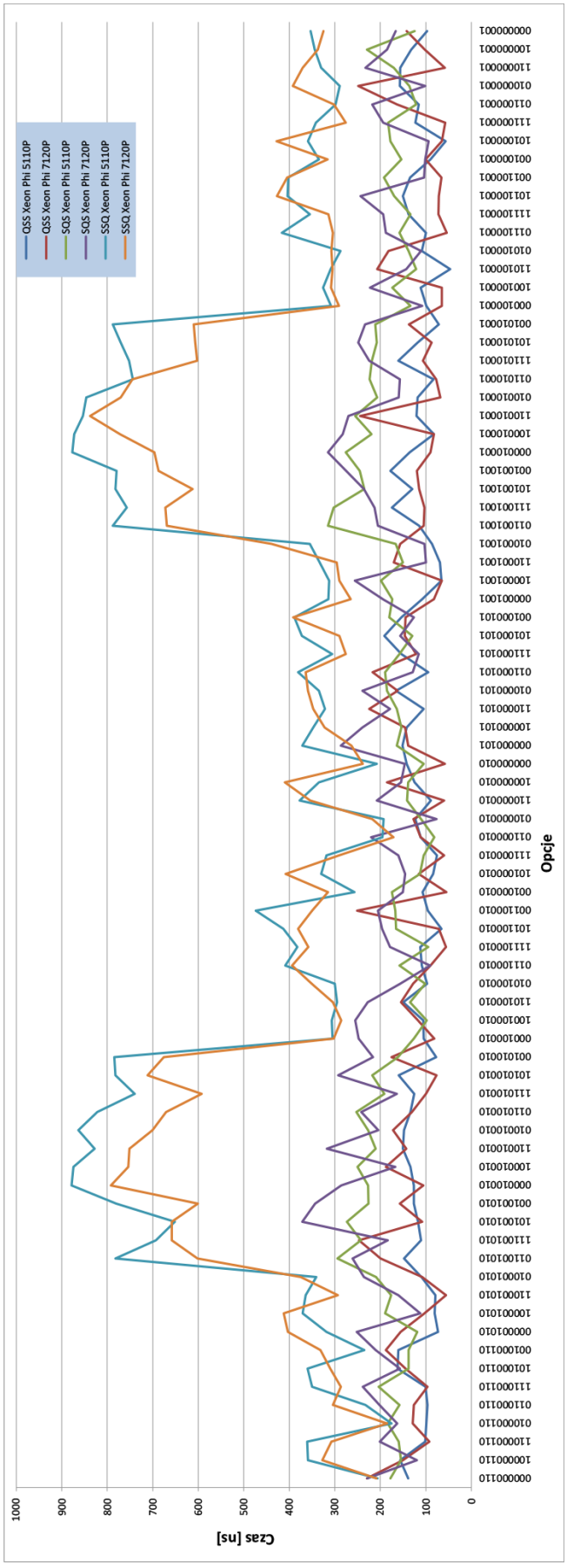
W przypadku koprocesorów Xeon Phi możemy zauważyć bardzo duże fluktuacje, powtarzające się względem opcji PADDING. Dla elementów czworościennych i zadania Poissona (Rys. C.1), obserwujemy większą stabilność wyników, z głównymi skokami wydajności związanymi z opcją CASFD. W tym przypadku jej wyłączenie redukuje liczbę obliczeń, gdyż są one wykonywane poza głównymi pętlami. W przypadku zadania konwekcji-dyfuzji i elementów czworościennych (Rys C.2), najlepsze wyniki uzyskujemy w przypadku minimalizacji użycia pamięci wspólnej, czyli w przypadkach gdy żadna z tablic nie jest w przechowywana, oraz w przypadku gdy jest w niej tylko najmniejsza tablica z danymi geometrycznymi. W przypadku elementów pryzmatycznych (Rys. C.3), jak w przypadku każdej z poprzednich architektur, największy spadek wydajności obserwujemy w algorytmie SSQ przy włączeniu redundantnych obliczeń pochodnych funkcji kształtu (CASFD). Pozostałe wyniki wykazują dosyć dużą spójność, aczkolwiek jak zauważono w rozdziale 5.4, odbiegają od wyników uzyskanych na innych architekturach.



Rysunek C.1: Automatyczny tuning parametryczny dla koprocesorów Intel Xeon Phi oraz zadania Poissona i elementów czworościennych



Rysunek C.2: Automatyczny tuning parametryczny dla koprocesorów Intel Xeon Phi oraz zadania konwekcji-dyfuzji i elementów czworościennych



Rysunek C.3: Automatyczny tuning parametryczny dla koprocesorów Intel Xeon Phi oraz zadania konwekcji-dyfuzji i elementów pryzmatycznych

Streszczenie

Głównym celem niniejszej pracy jest odpowiedź na pytanie jak efektownie realizować tworzenie macierzy sztywności w równoległych symulacjach Metody Elementów Skończonych z wykorzystaniem wielopoziomowej hierarchii pamięci i architektur masowo wielordzeniowych. W ramach tych badań, opracowano wydajną metodę programowania procedur aproksymacji, wykorzystywanych szeroko w zagadnieniach związanych z MES. Jako wzorcowa metoda została wybrana procedura całkowania numerycznego, ze względu na swoją nietrywialną strukturę oraz fakt, że jej optymalizacja jest często pomijana w badaniach naukowych na rzecz przyśpieszenia innych faz obliczeń MES. Dzięki swojemu skomplikowaniu i różnym wariantom, może się ona stać wzorem w jaki sposób należy realizować także inne algorytmy MES na architekturach takich jak akceleratory, procesory graficzne oraz procesory wielordzeniowe.

Praca ta zawiera pełną analizę badanego algorytmu oraz jego implementację na różne nowoczesne architektury takie jak procesory ogólnego przeznaczenia(CPU), karty graficzne (GPU), koprocesor Intel Xeon Phi oraz procesory hybrydowe IBM PowerXCell 8i i AMD Accelerated Processing Unit. W ramach badań, opracowano system automatycznego tuningu parametrycznego algorytmu całkowania numerycznego, dzięki czemu możliwym jest optymalne wykorzystanie wszystkich cech nowoczesnych architektur obliczeniowych, takich jak masowo wielordzeniowość, jednostki wektorowe czy wielopoziomowa hierarchia pamięci. Dzięki przeprowadzonym testom wskazano, jakie cechy nowoczesnych architektur procesorów, mają największy wpływ na wysoką wydajność obliczeń w algorytmach podobnych do badanego.

Wyniki uzyskane w ramach tej pracy dają wskazówki pomocne zarówno przy wyborze odpowiedniego sprzętu, jak i przy projektowaniu algorytmów go wykorzystujących.

Abstract

The main aim of this study is to answer the question: how to effectively implement the creation of the finite element stiffness matrix in parallel simulations of Finite Element Method using multi-level memory hierarchy and massively multicore architectures. In this work, author developed an efficient method of programming approximation procedures used widely in FEM. As an exemplary method, procedure of numerical integration was chosen, due to its non-trivial structure and the fact that its optimization is often ommited in research in favor of accelerating the other phases of FEM. Due to its complexity and different variants it can become the model of how to implement other FEM algorithms on such an architectures as accelerators, graphics processors and multicore processors.

This work contains a full analysis of the tested algorithm and its implementation on a variety of modern architectures, such as general purpose processors (CPU), graphics card (GPU), Intel Xeon Phi co-processor or heterogenous procesors IBM PowerXCell 8i and AMD Accelerated Processing Unit. During his study, author has developed a system for automatic parametric tuning of a numerical integration algorithm, making it possible to optimally use all the features of a modern computing architectures, such as massively multi-core, vector units or multi-level memory hierarchy. Thanks to the conducted tests, author indicated which characteristics of a modern CPU architectures, have the greatest impact on high performance computation in algorithms similar to numerical integration.

The results obtained during this study provide guidance for choosing the right equipment as well as choosing the optimal design of algorithms that use it.