AKADEMIA GÓRNICZO-HUTNICZA

IM. STANISŁAWA STASZICA W KRAKOWIE



Wydział Inżynierii Metali i Informatyki Przemysłowej



PROJEKT INŻYNIERSKI

pt.

„Modelowanie mikrostruktur materiałowych z wykorzystaniem automatów komórkowych na akceleratorach obliczeń GPGPU”

Imię i nazwisko dyplomanta: **Paweł Nowak**

Kierunek studiów: **Informatyka Stosowana**

Nr albumu: **232186**

Opiekun: dr inż. Łukasz Rauch

Podpis dyplomanta: Podpis opiekuna:

Kraków 2013

***Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszy projekt inżynierski wykonałem osobiście i samodzielnie i że nie korzystałem ze źródeł innych niż wymienione w pracy.***

Kraków, dnia ………….… Podpis dyplomanta…………….

*Serdeczne podziękowania dla mojej Kochanej Wioli, za wsparcie i motywację podczas pisania pracy. Podziękowania również kieruję do promotora dr inż. Łukasza Raucha, za pomoc merytoryczną, oraz cenne podpowiedzi i rady. Słowa podziękowań należą się również Arturowi Rodzajowi, za wsparcie techniczne związane z oprogramowaniem wykorzystanym podczas pisania niniejszej pracy.*

Spis treści

[1 WSTĘP 6](#_Toc345852784)

[2 WPROWADZENIE TEORETYCZNE 8](#_Toc345852785)

[2.1 ARCHITEKTURA I PROGRAMOWANIE NA GPU 8](#_Toc345852786)

[2.1.1 Architektura GPU 8](#_Toc345852787)

[2.1.2 OpenCL 9](#_Toc345852788)

[2.1.3 Budowanie i uruchamianie programów napisanych w OpenCL 12](#_Toc345852789)

[2.1.4 Grid Computing 14](#_Toc345852790)

[2.1.5 Projekt PL-Grid 15](#_Toc345852791)

[2.1.6 NVIDIA Tesla M-class 17](#_Toc345852792)

[2.2 WYDAJNOŚĆ OBLICZEŃ RÓWNOLEGŁYCH 19](#_Toc345852793)

[2.2.1 Przyspieszenie i efektywność 19](#_Toc345852794)

[2.2.2 Skalowalność równoległa 20](#_Toc345852795)

[2.3 AUTOMATY KOMÓRKOWE 20](#_Toc345852796)

[2.3.1 Definicja CA 20](#_Toc345852797)

[2.3.2 Zrównoleglone automaty komórkowe 21](#_Toc345852798)

[3 ZJAWISKA MIKROSTRUKTURALNE 22](#_Toc345852799)

[3.1 NAIWNY ROZROST ZIAREN 22](#_Toc345852800)

[3.1.1 Opis zjawiska 22](#_Toc345852801)

[3.1.2 Model naiwnego rozrostu ziaren do zastosować GPU 23](#_Toc345852802)

[3.1.3 Implementacja modelu 24](#_Toc345852803)

[3.2 REKRYSTALZACJA DYNAMICZNA 26](#_Toc345852804)

[3.2.1 Opis zjawiska 26](#_Toc345852805)

[3.2.2 Implementacja modelu 26](#_Toc345852806)

[3.2.3 Model rekrystalizacji dynamicznej do zastosowań GPU 30](#_Toc345852807)

[3.3 PĘKANIE ZMĘCZENIOWE 31](#_Toc345852808)

[3.3.1 Opis zjawiska 31](#_Toc345852809)

[3.3.2 Implementacja modelu 32](#_Toc345852810)

[3.3.3 Model pękania zmęczeniowego do zastosowań GPU 37](#_Toc345852811)

[4 WYNIKI 38](#_Toc345852812)

[4.1 Testy wydajnościowe zaimplementowanych modeli mikrostruktur materiałowych, przy różnej ilość procesorów GPU 38](#_Toc345852813)

[4.2 Porównanie obliczeń na CPU i GPU 46](#_Toc345852814)

[4.3 Przyspieszenie obliczeń na procesorach graficznych 47](#_Toc345852815)

[4.4 Skalowalność obliczeń na akceleratorach GPGPU 51](#_Toc345852816)

[5 WIOSKI 56](#_Toc345852817)

[6 BIBLIOGRAFIA 58](#_Toc345852818)

[7 Zawartość CD 59](#_Toc345852819)

[8 Spis ilustracji 60](#_Toc345852820)

[9 Spis tabel 61](#_Toc345852821)

[10 Spis wykresów 62](#_Toc345852822)

[11 Listing programów 63](#_Toc345852823)

[12 Dodatek: Implementacja modułu do debugowania źródła kodu, języka C99 dla OpenCL 64](#_Toc345852824)

# WSTĘP

Od powstania jednego z pierwszych komputerów ENIAC minęło już ponad 65 lat, możliwości obliczeniowe tej maszyny to 385 mnożeń lub ponad 5000 dodawań i odejmowań w ciągu sekundy[[1]](#footnote-1). Obliczenia o takiej wydajności pozwoliłyby, aby w przeciągu 80 tysięcy godzin pracy maszyna ta wykonała więcej obliczeń niż całą ludzkość od początku istnienia. Mimo że trudno w tej chwili jest oszacować wartość układu obliczeniowego podobnej klasy, to dla porównania wystarczy powiedzieć, iż ENIAC który zajmował powierzchnię 140 metrów kwadratowym, można obecnie zamknąć w rdzeniu o powierzchni 9 milimetrów kwadratowych. Współcześnie nastąpił olbrzymi postęp technologiczny i zgodnie z wciąż sprawdzającym się w praktyce prawe Moore’a liczba tranzystorów w pojedynczej kości układu scalonego podwaja się co 18 miesięcy. Przyrost ten daje możliwość do mnożenia etapów przetwarzania potokowego, liczby jednostek funkcjonalnych pracujących nad jednym strumieniem rozkazów i zwiększenia częstotliwości taktowania procesorów. Aby jeszcze zwiększyć wydajność stosuje się obliczenia równoległe, polegające na wykorzystywaniu szeregu procesorów, zdolnych do wspólnego rozwiązywania pewnego zagadnienia równoległego. Jakby jeszcze tego było mało, firma Nvidia w 2006 wprowadziła innowacyjną architekturę CUDA procesorów wielordzeniowych (głównie kart graficznych GPU), których moc obliczeniowa pozwala w wydajniejszy sposób rozwiązać problemy numeryczne, niż procesory sekwencyjne.

Obecna tendencja rozwoju architektury, a co za tym idzie oprogramowania, sprawia że programowanie równoległe zaczyna odgrywać podstawową rolę przy rozwiązywaniu problemów numerycznych. Dlatego też niniejsza praca inżynierska została poświęcona implementacji mikrostruktur materiałowych, w oparciu o automaty komórkowe, na akceleratorach obliczeń GPGPU. Do rozwiązania tego problemu, zastosowano architekturę OpenCL, która jest alternatywą dla CUDA. W przeciwieństwie do CUDA, która jest dostępna jedynie dla kart graficznych produkowanych przez firmę NVIDIA, OpenCL jest bardziej heterogeniczne i może być wykorzystywane na kartach graficznych innych producentów.

Przy pisaniu poniższej pracy została wykorzystana gotowa architektura, która zamyka w sobie sterowaniem procesorami graficznymi za pomocą OpenCL i służy jako interfejs do modelowanie zjawisk zachodzących w mikrostrukturze materiałów, zgodnie z ideą automatów komórkowych. Celem niniejszej pracy jest udowodnienie, iż odpowiednia implementacja modeli materiałowych opartych o automaty komórkowe na konwencjonalne procesory wielordzeniowe oraz procesory kart graficznych umożliwi zwiększenie efektywności działania tych algorytmów. Dzięki temu modele, i przeprowadzane za ich pomocą symulację, będą w stanie wierniej odwzorowywać zjawiska fizyczne, jakie zachodzą w rzeczywistym materiale. Jest to niezmiernie ważne, ponieważ rezultaty otrzymane metodami czysto teoretycznymi, w szeregu przypadkach wymagają potwierdzenia eksperymentalnego. Takie eksperymenty wiążą się z bardzo dużymi kosztami, a wymagania specjalistów w stosunku do dokładności przeprowadzanych badań wciąż rosną.

Niniejsza praca zawiera badania wydajności, w postaci przyspieszenia i efektywności obliczeń równoległych, przy zastosowaniu różnych konfiguracji sprzętowych. Obliczenia te zostały oparte na trzech modelach odwzorowujących zjawiska zachodzące w mikrostrukturach materiałowych. Są to: naiwny rozrost ziaren, rekrystalizacja dynamiczna i pękanie zmęczeniowe. Wszystkie te modele nie symulują wiernie procesów zachodzących w odpowiadających im zjawiskach, ponieważ nie jest to tematem niniejszej pracy. Zostały one zaimplementowane głównie z myślą o tym, aby wiernie odzwierciedliły złożoność obliczeniową dokładniejszych modeli.

W niniejszej pracy została opisana idea wykorzystania zastosowania procesorów wielordzeniowych. W jednym z rozdziałów przybliżono akceleratory obliczeniowe GPU TESLA™ M-class, na których zostały przeprowadzone badania dla niniejszej pracy. Jeden podrozdziałów poświęcony jest także projektowi plGrid, jako jedna z alternatyw dla obliczeń równoległych, jaką są obliczenia rozproszone. W części doświadczalnej pracy, skupiono się na zaimplementowanych modelach. Zestawiono je z rzeczywistymi modelami, przybliżono algorytmy, na których one się opierają, a szczególnie przybliżono reguły przejścia determinujące ich zachowanie.

Na podstawie zaimplementowanych modeli zbadano ich wydajność w poszczególnych konfiguracjach sprzętowych CPU-GPU, która została zestawiona na wykresach. Analizie zostały poddane czasy wykonywania poszczególnych modeli w oparciu o rozmiar przestrzeni automatów komórkowych, ilość wykorzystanych procesorów karty graficznej, a także w zależności od ilości procesorów strumieniowych. Wszystkie obliczenia zostały wykonane na akceleratorach graficznych GPU Tesla M2090.

# WPROWADZENIE TEORETYCZNE

## ARCHITEKTURA I PROGRAMOWANIE NA GPU

### Architektura GPU

Obliczenia GPU (Graphics Processing Unit) to użycie karty graficznej do uzyskania wyników obliczeń, zamiast tradycyjnie do renderowania grafiki. Procesory kart graficznych są wielordzeniowe i charakteryzują się wysoką wydajnością. Mogą zostać one stosowane do przyspieszenia różnorodnych zastosowań za pomocą przetwarzania równoległego. Duża programowalność kart graficznych daje ich użyteczność przy szybkich obliczeniach, poza oczywiście ich graficznymi możliwościami.

Narodziny rewolucji w obliczeniach, przy użyciu kart graficznych nastąpiły w listopadzie 2006 roku, kiedy to AMD wprowadziło na rynek CTM (Close To Metal), czyli nisko poziomowy interfejs programowania sprzętu. Umożliwia to programistom przeprowadzanie arytmetycznych obliczeń na procesorach kart graficznych, z wykorzystaniem ich pamięci. Technologie te nie spełniały jednak wszystkich pokładanych w nich oczekiwań klientów, ponieważ były mało uniwersalne – brak możliwości przeniesienia kodu z napisanego dla karty jednego producenta na komputer wykorzystujący kartę innego producenta. Dlatego też obecnie AMD ma na celu udostępnienie użytkownikom nowych możliwości obliczeń przy pomocy GPGPU.

GPGPU (ang. General-Purpose Computing on Graphics Processing Units) – ta metoda również daje możliwość użytkownikowi przeprowadzenia obliczeń przy użyciu kart graficznych. Metoda ta jest stosunkowo nowa, ponieważ dopiero karty graficzne zgodne z DirectX9, potrafiły wykonywać operacje na zmiennych zmiennoprzecinkowych o pojedynczej precyzji, w dodatku karty zostały wzbogacone w obsługę pętli i dynamicznych instrukcji warunkowych. Mimo wszystko karty graficzne dalej są projektowane specjalnie dla grafiki, a więc i są bardzo restrykcyjne, jeżeli chodzi o wykorzystanie ich do wykonywania operacji arytmetycznych. Ze względu na ich charakter są skuteczne tylko do problemów, które można rozwiązać stosując przetwarzanie strumieniowe, czyli SIMD[[2]](#footnote-2) (ang. Single Instruction, Multiple Data), a sprzęt może być używany tylko w określony sposób.

Idea GPGPU jest taka, że karty graficzne mogą przetwarzać nie tylko wierzchołki, ale mogą też przetwarzać wiele z nich równolegle. Jest to szczególnie istotne, gdy programista chce przetwarzać wiele wierzchołków lub fragmentów w ten sam sposób. W ten sposób została zastosowana architektura SIMD – procesory mogą działać równolegle, uruchamiając jedno jądro przez pojedynczy strumień rozkazów, dla wielu rekordów danych na raz. Rekordy danych wymagają podobnych obliczeń i zapewniają one równoległość, natomiast strumienie rozkazów to funkcje, które są stosowane, do każdego elementu danych. Ponieważ elementy procesorów graficznych są od siebie niezależne, nie ma możliwości używania danych współdzielonych czy statycznych. Dlatego każda dana musi zostać przeczytana z wejścia, muszą zostać wykonane na niej operacje, a następnie musi zostać ona zapisana do wyjścia. Dopuszczalne jest posiadanie wielu wejść i wyjść, ale nie można używać fragmentu pamięci, który jest przeznaczony zarówno do odczytu, jak i zapisu.

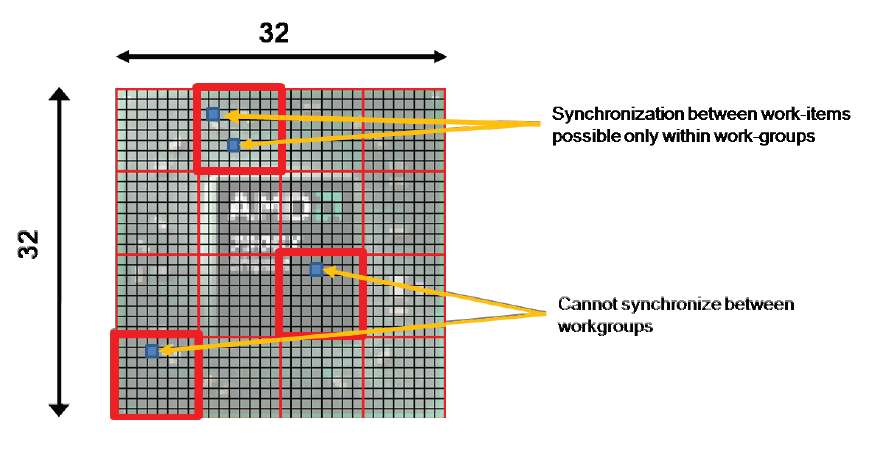
### OpenCL

Open Computing Language (OpenCL) jest platformowym standardem dla programowania komputerów składających się z kombinacji CPU, GPU i innych procesorów. Takie platformy zwane są heterogeniczne, a ich znaczenie znacząco wzrosło na przełomie ostatnich kilku lat. OpenCL jest pierwszym standardem, który bezpośrednio zaspokaja potrzeby rynkowe. W dodatku jest on otwartym i nieodpłatnym produktem.

Przy pomocy OpenCL można pisać pojedyncze programy, które mogą być uruchomione na wielu systemach, począwszy od telefonów komórkowych, poprzez laptopy i skończywszy na superkomputerach. Żadne inne programowanie równoległe nie jest tak przenaszalne, co jest powodem, dla którego OpenCL, odgrywa tak ważną rolę i ma potencjał do zrewolucjonizowania runku IT.

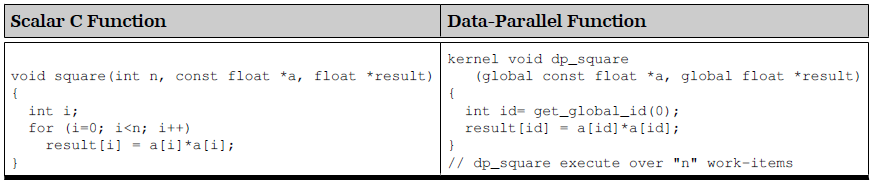
OpenCL zapewnia tak wysoką przenaszalność, poprzez wystawienie sprzętu, a nie ukrywaniem go za eleganckim modelem abstrakcyjnym. To oznacza, że programista jasno musi określić platformę, jej kontekst i zaplanować pracę na różnych urządzeniach.

OpenCL umożliwia również grupowanie elementów pracy razem w „grupy robocze”, jak pokazano na rysunku 2.1. Rozmiar każdej grupy roboczej jest definiowana przez jej własny indeks w przestrzeni. Wszystkie elementy robocze w tej samej grupie roboczej, są wykonywane razem na tym samym urządzeniu. Powodem dla którego wykonywane są one na tym samym urządzeniu, jest współdzielenie lokalnej pamięci i synchronizacja. Globalnie elementy robocze są niezależne i nie mogą być synchronizowane. Synchronizacja jest tylko dozwolona pomiędzy elementami roboczymi w tej samej grupie.



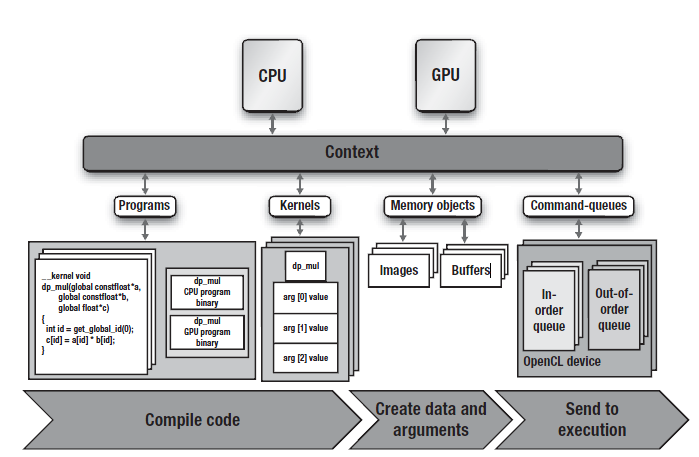
**Rysunek 2.1 Praca elementów roboczych w obrębie tej samej grupy i poza nią.[2]**

Takie rozwiązanie jak pokazane na rysunku 2.1 wymaja od programisty nowego podejścia, różniącego się od przeciętnego programowania proceduralnego. Przykład można zobaczyć na rysunku 2.2.



Rysunek 2.2 Programowanie proceduralne vs. programowanie równoległe. [2]

Platforma OpenCL jest dzielona na dwa interfejsy programistyczne aplikacji (API): platformy i wykonywalny. API platformy, ponieważ umożliwia aplikacji tworzenia listy urządzeń obsługujących OpenCL i zarządzania nimi poprzez tak zwany „context”. Natomiast wykonywalne API daje możliwość użycia context’u do uruchomienia (w ty kompilacji i budowania) źródła docelowego programu na urządzeniach obsługujących OpenCL. Poniższy rysunek 2.3, obrazuje podstawowe kroki podczas tworzenia dowolnego programu napisanego w OpenCL.



Rysunek 2.3 Etapy tworzenia programy napisanego przy pomocy OpenCL.[1]

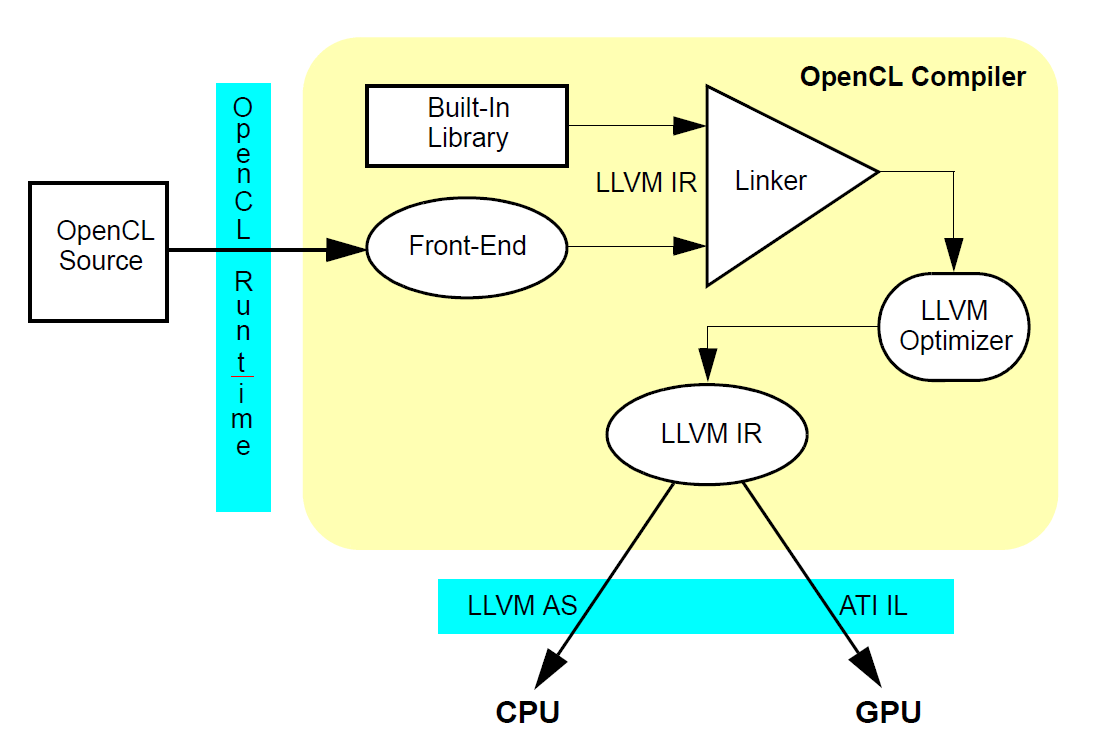
Aby uruchomić program korzystający z biblioteki OpenCL należy postępować zgodnie z następującymi krokami:

1. Stworzenie listy platform na których może zostać uruchomiona aplikacja napisana przy pomocy OpenCL, a także listy urządzeń przez które ta aplikacja będzie wykonywana.
2. Stworzenie context’u do kontroli nad urządzeniami do korzystania z urządzeń do obsługi OpenCL.
3. Napisanie programu, który będzie uruchomiony na jednym lub kilku urządzeniach.
4. Wybór źródła do wykonania.
5. Przypisanie pamięci w programie „gospodarzu”[[3]](#footnote-3), lub na urządzeniach.
6. Kopiowanie pamięci danych do urządzeń, o ile to potrzebne.
7. Przesłanie argumentów do źródła funkcji.
8. Przypisanie źródeł do kolejki komend, w celu ich wykonania.
9. Kopiowanie wyników z urządzeń obliczeniowych, do programu gospodarza.
10. Zwalnianie zalokowanej pamięci.

Specyfikację do OpenCL można pobrać z [www.khronos.org/opencl](http://www.khronos.org/opencl), można się tam nauczyć wszystkich konstrukcji tej architektury. Standard ten został opublikowany 8 grudnia 2008 roku. Jest on stworzony i rozwijany przez konsorcjum Khronos Group zrzeszające wiele gigantów branży IT, odpowiedzialne również za takie technologie jak OpenGL czy OpenAL. Apple było pomysłodawcą frameworku OpenCL, oraz do tej pory jest właścicielem marki. Od samego początku współpracowały również takie firmy jak Intel, AMD, IBM oraz Nvidia. Producenci tych urządzeń dostarczają odpowiednie sterowniki obsługujące interfejs OpenCL.

### Budowanie i uruchamianie programów napisanych w OpenCL

Aplikacja OpneCL zawiera program gospodarza (nazywany z ang. host’em), napisany w języku C/C++ i opcjonalny program źródłowy (z reguły zapisywany z rozszerzeniem .cl). Do skompilowania całej aplikacji (składającej się z gospodarza i jądra), najpierw należy skompilować program gospodarza. Może to być zrobione za pomocą ogólno dostępnych kompilatorów, takich jak g++ albo MSCV++. Natomiast jądro aplikacji OpenCL, musi zostać skompilowane przez kompilator OpenCL, wewnątrz urządzenia za pomocą specjalnych jego plików binarnych.



Rysunek 2.4 „Zestaw narzędziowy” kompilatora OpenCL [4].

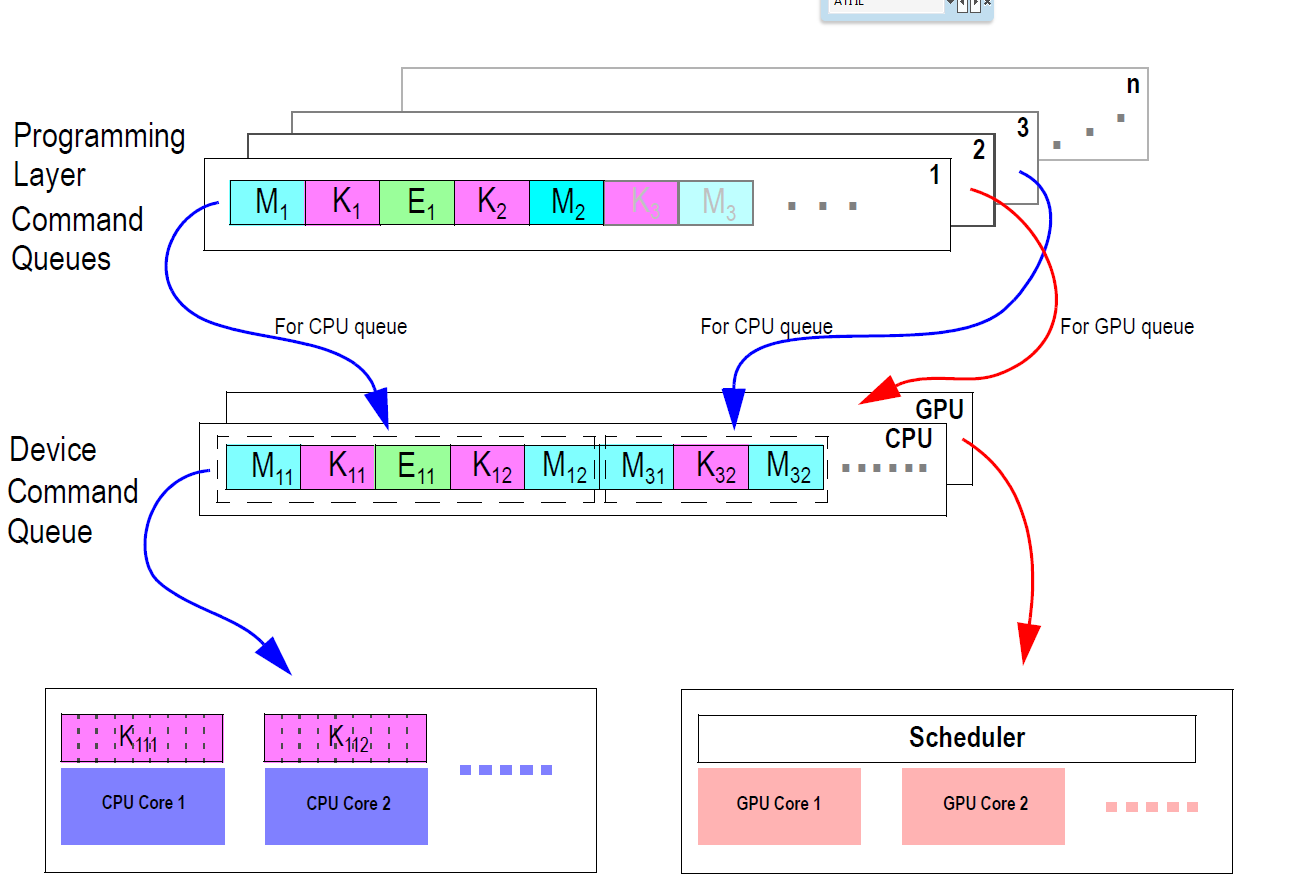
Kompilator ten używa nakładki do standardowego języka C, oraz platformę nisko poziomowej maszyny wirtualnej (LLVM), z rozszerzeniami OpenCL. Kompilator rozpoczyna pracę ze źródłem OpenCL, tak że program użytkownika przechodzi przez interfejs środowiska uruchomieniowego OpenCL (Rysunek 2.4). Następnie wcześniej wspomniana nakładka tłumaczy źródło OpenCL na kod LLVM IR. Dzięki temu OpenCL utrzymuje specyficzne informacje, takie jak struktury metadanych[[4]](#footnote-4) (np. do debugowania jądra, nakładka tworzy struktury meta danych do przetrzymywania informacji o debugowaniu). Nakładki wspierają dodatkowe typy danych (takie jak np. int4, float8, itp.), dodatkowe słowa kluczowe (get\_globalid(), barier(), itp.). Ponadto wykonuje dodatkowe składniowe i semantyczne kontrole, w celu sprawdzenia czy jądro spełnienia warunki specyfikacji OpenCL. Danymi wejściowymi dla linkera LLVM, są dane wyjściowe z nakładek kompilatora i biblioteki ze zbudowanymi funkcjami. W ten sposób wiązane są zbudowane funkcje OpenCL z jądrem i te powiązane dane przesyłane są do optymalizatora LLVM IR.

Dla przetwarzań procesora, środowisko uruchomieniowe OpenCL używa LLVM[[5]](#footnote-5) AS aby wygenerować pliki binarne x86. Środowisko uruchomieniowe automatycznie określa liczbę elementów przetwarzanych, lub rdzeni obecnych w CPU i dystrybuuje jądro OpenCL pomiędzy je.

Do obliczeń GPU, środowisko uruchomieniowe OpenCL, za pomocą modułu obliczeń warstwy abstrakcyjnej (CAL) języka IL, uruchamia niekompletny język przetwarzania obrazów (post-processes) ATI IL[[6]](#footnote-6) i generuje z niego kompletny język ATI IL. Proces ten dodaje makra z bazy danych specyficzne dla GPU. Wtedy warstwa środowiska uruchomieniowego OpenCL usuwa niepotrzebne funkcje i przesyła kompletny język IL do kompilatora CAL, który przeprowadza kompilację, w wyniku której otrzymywane są specyficzne pliki binarne dla GPU.

Środowisko uruchomieniowe systemu przydziela prace w kolejkach poleceń do podstawowych urządzeń (Rysunek 2.5). Rozkazy są umieszczane w kolejce używając poleceń typu *clEnqueue*. Ogólnikowo komendy uruchomieniowe jądra OpenCL mogą być sklasyfikowane w trzech kategoriach:

* Komendy jądra (np. *clEnqueueNDRangeKernel()*)
* Komendy zarządzania pamięcią (np. *clEnqueueReadBuffer()*)
* Komendy obsługi zdarzeń (np. *clEnqueueWaitForEvents()*)



Rysunek 2.5 Środowisko uruchomieniowe OpenCL [4].

### Grid Computing

Grid Computing to dyscyplina w dziedzinie IT, która wykorzystuje możliwości aktualnej sieci komputerowej, zwanej Internetem w której może być połączona nieograniczona ilość urządzeń, w tak zwaną siatkę (ang. grid), do obliczeń wysokiej mocy. To nowe innowacyjne podejście do obliczeń może być łatwo osiągalne, dzięki siatce o wielkiej mocy energetycznej, która jest dostarczana do naszych domów i wielkich przedsiębiorstw codziennie. Taka siatka może być interpretowana, jako rozproszony system z nieinteraktywnymi obciążeniami, które wymagają dużej liczby plików. Rozwiązanie to używa standardowych otwartych protokołów i interfejsów ogólnego przeznaczenia, oraz dostarcza usług odpowiedniej jakości. Rozwiązanie to jest podobne do klastrów komputerowych[[7]](#footnote-7), jednak to nie jest to samo. To co odróżnia Grid od konwencjonalnych systemów wysokiej mocy obliczeniowej, takiej, jak np. właśnie klastry komputerowe, to że siatka urządzeń jest bardzo luźna, heterogeniczna i rozproszona geograficznie. Chociaż pojedynczy Grid z reguły jest przeznaczony do określonego zastosowania, powszechnie stosowany jest w sieci do różnych celów.

Za początki Gridu uważa się internetowy projekt obliczeń rozproszonych SETI@home (ang. Search for Extra-Terrestrial Intelligence), którego celem było poszukiwanie w kosmicznym szumie radiowym sygnałów od pozaziemskich cywilizacji. Przeciętny użytkownik komputera mógł pobrać fragment danych (sygnałów odbieranych przez radioteleskopy) na swój sprzęt i na nim to właśnie przeprowadzić obliczenia, wykorzystując wolne cykle CPU. To nowatorskie podejście do problemu pozwoliło na równoczesne wykorzystanie milionów komputerów rozproszonych po całym świecie.

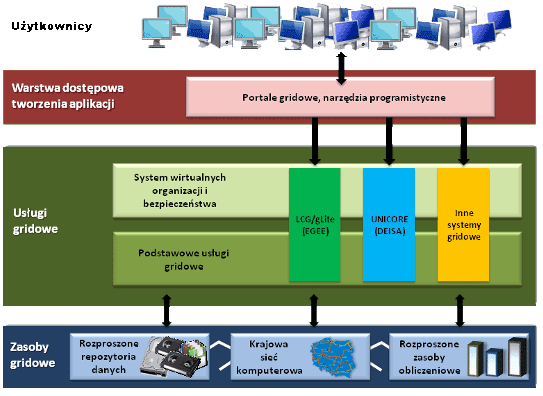
Grid jest formą rozproszonych obliczeń, w których „wirtualny super komputer” składa się z wielu komputerów w sieci luźno powiązanych i działających wspólnie wykonujących duże zadania. Dla pewnych zastosować Grid, bądź obliczenia rozproszone, mogą być postrzegane jako specjalnego rodzaju obliczenia równoległe, które odbywają się na kompletnych komputerach (ze zintegrowanym CPU, wspólna pamięcią, zasilaniem, itp.), podłączonych do sieci (prywatnych, publicznych lub do Internetu), za pomocą konwencjonalnego interfejsu sieciowego, takiego jak Ethernet. Odróżnia to Grid od tradycyjnego pojęcia superkomputer, który ma wiele procesorów powiązanych ze sobą za pomocą szybkiej magistrali.

### Projekt PL-Grid

W okresie od początku 2009 roku do pierwszego kwartału 2012 w Polsce był realizowany projekt PL-Grid. Ma on na celu dostarczenie polskiej społeczności naukowej usług informatycznych opartych na gridowych klastrach komputerowych, służących e-Science w różnych dziedzinach. W wyniku projektu na terenie Polski powstała otwarta infrastruktura gridowa wspierająca naukę, poprzez integrację danych doświadczalnych i wyników zaawansowanych symulacji komputerowych. Cel projektu udało się zrealizować uzyskując moc obliczeniową 230.16 TFlops’ów, z pamięcią operacyjną 51.33 TB, oraz przestrzenią dyskową o rozmiarze 3.6 PB, przy 2316 rdzeniach. Beneficjentem całego przedsięwzięcia jest Akademickie Centrum Komputerowe Cyfronet AGH. System jest skalowalny i pozwala na dołączenie lokalnych klastrów komputerowych uczelni, instytutów badawczych czy „platform technologicznych”. Infrastruktura jest kompatybilna i interoperabilna z gridem europejskim i światowym. Projekt zapewnia również wygodny dostęp do zasobów komputerowych dla zespołów badawczych spoza środowisk, w których działają centra KDM. Organizacyjnie projekt jest podzielony na zadania, koordynowane przez poszczególnych partnerów konsorcjum. W skład projektu wchodzą:

* Akademickie Centrum Komputerowe CYFRONET AGH w Krakowie
* Interdyscyplinarne Centrum Modelowania Matematycznego i Komputerowego w Warszawie
* Instytut Chemii Bioorganicznej PAN - Poznańskie Centrum Superkomputerowo Sieciowe w Poznaniu
* Centrum Informatyczne Trójmiejskiej Akademickiej Sieci Komputerowej w Gdańsku
* Wrocławskie Centrum Sieciowo - Superkomputerowe we Wrocławiu

Struktura projektu obejmuje bazę, ze specjalizowanymi systemami dla różnych dziedzin nauki, wraz z usługami i narzędziami dostosowanymi do wykorzystania tych aplikacji.



Rysunek 2.6 Struktura projektu PL-Grid[[8]](#footnote-8).

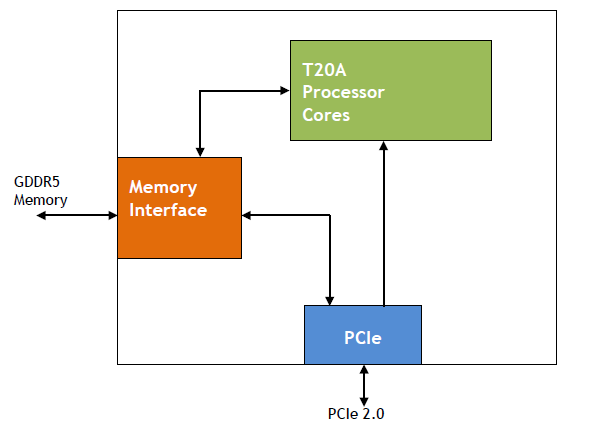
Oprogramowanie e-infrastruktury gridowej obejmuje:

* narzędzia użytkownika,
* biblioteki programistyczne,
* system wirtualnych organizacji,
* system zarządzania danymi,
* system zarządzania zasobami

Dostęp do infrastruktury gridowej oferowanej w ramach projektu PL-Grid jest darmowy dla naukowców i wszystkich osób prowadzących działalność naukową, związaną z uczelnią lub instytutem naukowym w Polsce. Aby zostać użytkownikiem projektu należy zarejestrować się w Portalu Użytkowników Infrastruktury PL-Grid[[9]](#footnote-9).

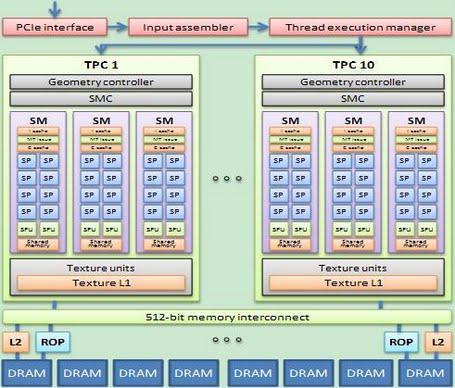
### NVIDIA Tesla M-class

Akceleratory obliczeniowe GPU TESLA™ M-class - czyli najszybsze procesory równoległe do zastosować naukowych o wysokiej wydajności (HPC). Karty te dzięki swojej wysokiej wydajności znalazły zastosowanie przy badaniach sejsmicznych, symulacjach biochemicznych, pogodowych i modelowaniu klimatu, przetwarzaniach sygnałów, obliczeniach finansowych, fizyce i chemii kwantowej, analizie danych, a także przy komputerowych wspomaganiu prac inżynierskich. Ponadto nowy układ ma znaleźć zastosowanie w szerokiej gamie programów do obliczeń wielkoskalowych, akcelerowanych przez procesor graficzny. Wśród nich znajdą się m.in. aplikacje do obliczeń dynamiki molekularnej, NAMD i GROMACS; aplikacje CAD, Ansys Mechanical, Altair Acusolve i Simulia Abaqus Do obliczeń w niniejszej pracy inżynierskiej został zastosowany model procesora M2090. Omawiane akceleratory GPU M-class zostały oparte na architekturze CUDA™, o nazwie kodowej procesora graficznego „Fermi”. Seria 20 procesorów graficznych GPU, jako pierwsza dostarczyła większą od serii 10 wydajność obliczeń zmiennoprzecinkowych, o podwójnej precyzji z czterordzeniowym procesorem x86 i jako pierwsza była wyposażona w pamięć ECC. Oparty na architekturze Fermi model procesora M2090 dostarcza od 520 do 640 Gigaflops’ową wydajność[[10]](#footnote-10) (szczytowa wartość to 665) obliczeń zmiennoprzecinkowych, o podwójnej precyzji i 1331 Gigaflops’ową wydajność o pojedynczej precyzji. Przepustowość 6 GB pamięci GDDR5, przy wyłączonych mechanizmach ECC to 177 GB/s. Procesor ten liczy sobie 512 rdzeni CUDA i jest jedynym produktem NVIDIA, który wyposażony jest w dwa silniki DMA umożliwiające dwukierunkową komunikację z magistralą PCIe, jak to zostało przedstawione na rysunku 2.7.



Rysunek 2.7 Schemat blokowy używanego w TESLA™ M2090podwójnego gniazda modułu obliczeniowego [9].

Architektura Tesla bazuje na skalowalnej tablicy procesorów. Rysunek 2.8 pokazuje schemat blokowy procesora graficznego GeForce 8800 ze 128 procesorami strumieniowymi SP (z j. ang. streaming - processor), zgromadzone w 16 rdzeni strumieniowych wieloprocesorów SMs (z j. ang streaming multiprocessors), które to natomiast znajdują się w ośmiu niezależnych jednostkach przetwarzania, nazywanych klastrami tekstur/procesorów TPCs (z j. ang. texture/processor clusters). Przepływ pracy rozpoczyna się od góry do dołu, począwszy od interfejsu gospodarza z systemem PCIExpress.



Rysunek 2.8 Schemat blokowy używanego w TESLA™ M2090 podwójnego gniazda modułu obliczeniowego, na podstawie procesora graficznego GeForce 8800 [15].

## WYDAJNOŚĆ OBLICZEŃ RÓWNOLEGŁYCH

### Przyspieszenie i efektywność

Programowanie CPU i GPU przy pomocy architektury OpenCL, ma podobnie zamierzenia, ja już stosunkowo stare technologie obliczeniowe, takie jak na przykład Pthread, czy OpenMP. Wszystkie te metody obliczeniowe, mają za zadanie zrównoleglić kody, tak by zwiększyć wydajność aplikacji, która rozumiana jest jako: skrócenie czasu działania programów, zwiększenie wydajności obliczeń, uzyskanie szybszego przetwarzania.

Wydajność można wyrazić jako liczbę operacji na sekundę – MIPS (Million Instruction Per Second), lub liczbę operacji zmiennoprzecinkowych na sekundę – MFLOPS (FLoating point Operation Per Second). Przy czym należy pamiętać, że niemożliwe jest oszacowanie jak szybko wykonywana jest pojedyncza instrukcja (w tym zmiennoprzecinkowa). Wyróżnia się dwie miary względne wydajności obliczeń równoległych, w których porównuje się czasy wykonywania programu na jednym i wielu procesorach. Pierwszą z nich jest przyspieszenie obliczeń, które wyraża się wzorem 2.1 i w którym to czas rozwiązania zadania najlepszym algorytmem sekwencyjnym na pojedynczym procesorze, a to czas rozwiązania zadania rozważanym algorytmem równoległym na p procesorach. W praktyce często zamiast używa się .

(2.1)

Drugą miarą wydajności jest efektywność zrównoleglenia, którą wyraża wzór 2.2. Idealną sytuacją jest uzyskanie liniowego przyspieszenia i 100% efektywności.

(2.2)

Przyspieszenie można zmierzyć dzięki analizie, jak dla danej liczby procesorów przyspieszenie obliczeń równoległych jest większe, dla zadań większych, z założeniem że rozpatrywany jest ten sam problem, ten sam algorytm, dla tego samego programu, a zmienia się tylko wielkość zadania. Przyspieszenie obliczeń równoległych jest większe, jeżeli dla zadań, dla których przy stałej liczbie procesorów, ilość obliczeń rośnie szybciej niż narzut obliczeń równoległych. Jeżeliby wprowadzić pojęcie pracy, jako liczby operacji wykonywanych przy realizacji zadania, to natomiast przyspieszenie i efektywność można wyrazić jako funkcję dwóch parametrów – liczby procesorów i wielkości zadania S(p, W) i E(p, W). Dzięki temu zabiegowi otrzymuje się funkcję dwóch zmiennych, którą można dalej analizować rozpatrując różne jej przekroje. Istnieje pojęcie przyspieszenia skalowanego, czyli przyspieszenia jakie uzyskuje się rozważając dla danej liczby procesorów p zadanie o rozmiarze p-krotnie większym od zadania rozwiązywanego na pojedynczym procesorze. Jako funkcja liczby procesorów przyspieszenie skalowane jest równe:

(2.3)

Liniowe przyspieszenie skalowalne uzyskuje się wtedy kiedy czas rozwiązania nie zmienia się przy rozwiązaniu zadania o rozmiarze p-krotnie większym na p razy więcej procesorach – efektywność zrównoleglenia jest wtedy stała.

### Skalowalność równoległa

Innym pojęciem, który określa zachowanie programu równoległego dla rosnącego rozmiaru zadania i rosnącej liczby procesorów, to skalowalność równoległa. Program jest skalowalny równolegle, jeżeli istnieje taka funkcja W(p), że efektywność obliczeń równoległych jako funkcja E(W(p), p), jest ograniczona od dołu przez liczbę większą od zera. Jeśli W(p) jest funkcją liniową, to mówimy o liniowej skalowalności.

Przy zapewnieniu wysokiej wydajności obliczeniowej jest to, aby programista zwracał szczególną uwagę na złożoność algorytmu, jako jedną z najważniejszych cech charakteryzujących algorytm. Podstawowymi zasobami systemowymi uwzględnianymi w analizie algorytmów, są czas działania oraz obszar zajmowanej pamięci. Złożoność określa skalowalność, czyli zachowanie algorytmu w obliczu zmian rozmiaru rozwiązywanego problemu i często jest mylona z wydajnością algorytmu (mierzonej zużyciem czasu procesora, zajętością pamięci operacyjnej, oraz wykorzystaniem pamięci dyskowej). Przykładowo, gdy powiemy że dany algorytm jest w stanie przetworzyć 1000 rekordów w czasie ok. 10 milisekund nie wiele mówi o złożoności tego algorytmu, bo ten czasu zależy od wielu czynników poza samym algorytmem np. szybkością procesora. Ważne w złożoności jest to jak ten czas zmieni się gdy liczba przetwarzanych rekordów będzie kilkukrotnie większa. Liczba operacji wcale nie musi być znana dokładnie, istotne jest jak zmienia się ona w skutek zmian rozmiaru rozwiązywanego problemu. Podczas analizy problemu dla programisty najważniejsza jest złożoność czasowa.

## AUTOMATY KOMÓRKOWE

### Definicja CA

Za twórcę automatów komórkowych (ang. Cellular Automaton) uważa się Johna von Neumanna, węgierskiego matematyka, inżyniera, chemika, fizyka i informatyka, który pracował głównie w Stanach Zjednoczonych. Głównym jego celem było stworzenie maszyny samosterującej, która replikowałaby swoją budowę i przekazywała dalej. Automaty komórkowe są strukturą określone przez *n* wymiarową siatkę komórek. Każdy skończony zbór komórek posiada swój stan, który może ulec zmianie, jest to tak zwana przestrzeń automatów. W definicji automatów komórkowych istnieje również pojęcia otoczenia, które określa najbliższych sąsiadów danej komórki za pomocą promienia i rodzaju sąsiedztwa. Otoczenie może być rozpatrywane w przestrzeni 1D, 2D, 3D i nD. Zmiany stanów komórek w kolejnej iteracji są determinowane, przez reguły przejścia, które ściśle zależą od stanów sąsiednich komórek oraz od aktualnego stanu komórki. Zazwyczaj zasada aktualizacji stanu komórki jest taka sama dla każdej komórki, nie zmienia się w czasie i stosuje się do całej sieci jednocześnie, choć są znane takie wyjątki, jak probabilistyczne i asynchroniczne automaty komórkowe. Przy różnego rodzaju symulacjach ważne jest aby poprawnie określić warunki brzegowe. Istnieją następujące typy warunków brzegowych:

* periodyczne – przykładowo symulując cząstkę gdy dotrze do krawędzi w kolejnym kroku pojawi ona się z drugiej strony; sąsiad danej komórki przy krawędzi znajduje się po drugiej stronie;
* zamknięte pochłaniające – brzegi siatki wypełnione są z góry ustalonymi wartościami;
* zamknięte odbijające – brzegi siatki tworzą swego rodzaju barierę od której funkcje przejścia się odbijają wnosząc ponownie swój wkład do zmieniających się stanów komórek w przestrzeni automatów komórkowych.

Ważnym elementem jest dobór odpowiednich warunków początkowych czyli stanów komórek w iteracji zerowej. Od tych ustawień zależy zachowanie automatu podczas ewolucji a tym samym uzyskanie poprawnych wyników symulacji. Niejednokrotnie liczba możliwych funkcji przejścia jest bardzo duża, co prowadzi do znacznego wydłużenia czasu obliczeń. Aby ograniczyć ten czas możliwe jest wykorzystanie algorytmów selekcji zbioru reguł zapewniających wymagany wynik końcowy.

### Zrównoleglone automaty komórkowe

Zrównoleglenie automatów komórkowych niesie za sobą ograniczenia, związane z tym, że każda kolejna przestrzeń, wygenerowana w kolejnym kroku czasowym, jest zależna od przestrzeni z poprzedniego kroku czasowego. Jest to bardzo duże ograniczenie, ponieważ efektywność zrównoleglenia nie może zostać w pełni wykorzystana. Wszystkie dane będące wynikiem operacji wykonywanych na przestrzeni automatów komórkowych muszą, zostać ze sobą zsynchronizowane przed przystąpieniem do kolejnego kroku czasowego. Narzuty koszty czasowe pochodzą również z kosztu podziału przestrzeni pomiędzy rdzenie, a później procesory strumieniowe kart graficznej.

Cała przestrzeń automatów komórkowych dzielona jest na mniejsze podprzestrzenie, które następnie przekazywane są odpowiednio docelowemu procesorowi strumieniowemu, który wykonuje obliczenia na tak zwanych (z j. ang.) work item’ach. Rdzenie procesorów graficznych natomiast dostają do swojej dyspozycji przestrzenie automatów, zwane grupami roboczymi, w obrębie których to każdy work item, może że się ze sobą komunikować. Wszystkie elementy robocze w tej samej grupie roboczej, są wykonywane razem na tym samym urządzeniu, ponieważ współdzielą pamięć i są synchronizowane ze sobą.

Należy też pamiętać, że operacje które są przekazywane procesorom strumieniowym w przypadku obliczeń w oparciu o automaty komórkowe, to przede wszystkim reguły przejścia, które są implementowane, za pomocą instrukcji sterujących, takich tak na przykład if (…) {…} else {…}, a których to złożoności obliczeniowe są bardzo małe.

# ZJAWISKA MIKROSTRUKTURALNE

## NAIWNY ROZROST ZIAREN

### Opis zjawiska

Rozrost ziarna nazywamy proces metaloznawstwa polegający na zwiększaniu się rozmiaru krystalitu (ziarna) w materiale na skutek wysokiej temperatury. Proces ten następuje gdy rekrystalizacja jest skończona i dalsze zmniejszanie energii wewnętrznej może być osiągnięte jedynie przez zmniejszenie powierzchni granicy ziarna. Termin rozrostu ziaren jest powszechnie używany w metalurgii, ale używa się go także w odniesieniu do ceramiki i minerałów.

Większość materiałów wykazują efekt Halla Petch w temperaturze pokojowej, a więc wykazują wyższą granicę plastyczności, gdy wielkość ziarna jest mniejsza. W wysokich temperaturach jest odwrotnie, za sprawą otwartego, nieuporządkowanego charakteru granic ziaren, który oznacza że ​​wakansy[[11]](#footnote-11) mogą dyfundować szybciej w dół granic, prowadząc do szybszego pełzania. Od momentu kiedy granice są regionami wysokich energii, stanowią doskonałe obszary dla zarodkowania innych faz.

Wyróżniamy dwa podstawowe rodzaje rozrostu: naturalny i anormalny. Naturalny rozrost polega na ciągłym wzroście ziaren w taki sposób, że ich wielkość rośnie jednakowo w całej objętości materiału. Natomiast anormalny rozrost ziarn (rekrystalizacja wtórna) – polega na wzroście niewielkiej liczby dużych ziarn kosztem innych, których wielkość nie ulega zmianie.

### Model naiwnego rozrostu ziaren do zastosować GPU

Implementacja naiwnego rozrostu ziaren nie jest trudnym zadaniem. Warto przyjrzeć się jednak niektórym jej aspektom, takim jak na przykład weryfikacja czy ziarno znajduje się na granicy, z uwzględnieniem, że przestrzeń jest interpretowana przez procesory strumieniowe karty graficznej i jest podzielona, przez co komórki w rzeczywistości nie leżą obok siebie. Metoda ta przydaje się później, przy implementacji rekrystalizacji dynamicznej, w przypadku określania czy sąsiad danej komórki zrekrystalizował w poprzednim kroku czasowym.

for(int i = 0; i< NUM\_NEIGH; i++){

temp = getNeighbourhood(gx, gy, i);

if( CELL\_GRAIN(temp.x, temp.y) != 0 ){

for(int j = 0; j < amountNgh; j++){

if( NghNo[j] ==

CELL\_GRAIN(temp.x, temp.y) ){

NghAmount[j] ++;

newGrain = 0;

}

}

if ( newGrain == 1 ){ NghNo[amountNgh++] =

CELL\_GRAIN(temp.x, temp.y );

}

}

newGrain = 1;

}

Listing kodu 3.1 Fragment kodu jądra programu naiwnego rozrostu ziarna,

odpowiedzialny za weryfikację sąsiadów rozpatrywanej komórki

Zmienna globalna NUM\_NEIGH, określa liczbę sąsiadów rozpatrywanej komórki w danym sąsiedztwie i tak na przykład dla sąsiedztwa heksagolanego wynosi ona sześć. Każdy z sąsiadów za pomocą instrukcji sterującej for jest weryfikowany, przy pomocy funkcji getNeighbourhood, która zwraca na podstawie współrzędnych rozpatrywanej komórki, przesłanych przez wartość do argumentów funkcji i numery sąsiada *i* współrzędne tego sąsiada. Obiekt tmp, który jest wbudowanego typu uint2 zgodnego ze specyfikacją OpenCL, zawiera współrzędne sąsiada. Te współrzędne zapewniają dostęp za pomocą funkcji CELL\_GRAIN, do id danej komórki, a funkcje CELL\_GRAIN\_INT i CELL\_GRAIN\_REAL, dają dostęp odpowiednia do wektora zmiennych typu całkowitego i zmiennoprzecinkowego pojedynczej precyzji.

### Implementacja modelu

Naiwny rozrost ziaren jest metodą automatów komórkowych, która ma na celu uzyskanie przestrzeni. Każda komórka tej przestrzeni ma przypisane tylko jedno ziarno. Taka przestrzeń może zostać później wykorzystana, do symulacji innych procesów metalurgicznych, jak na przykład rekrystalizacji dynamicznej czy pękania zmęczeniowego. Zjawisko to posiada cechy probabilistyczne, ponieważ zarodki są rozmieszczane losowo w przestrzeni. Ilość zarodków powinna zostać dobrana, przed ich rozlokowaniem. Naiwny rozrost ziaren posiada wszystkie cechy automatów komórkowych. Dobór warunków brzegowych jest dobrowolny i może być dobrany do stawianego problemu. W niniejszej pracy zostały wykorzystane periodyczne warunki brzegowe, co oznacza, że jeśli ziarno wychodzi poza obszar automatów, to jego kontynuacja znajduje się na przeciwległym brzegu.



a.)

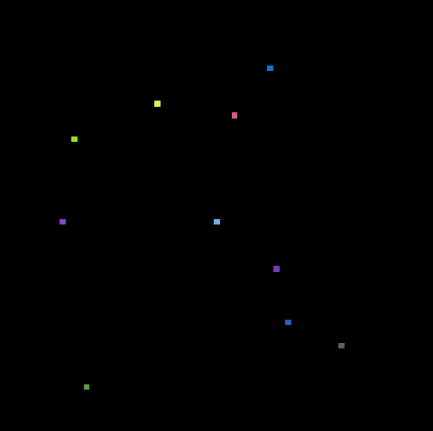
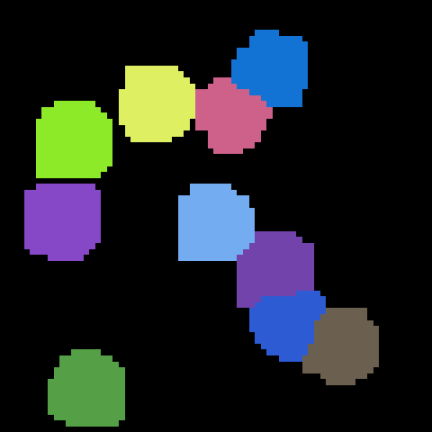
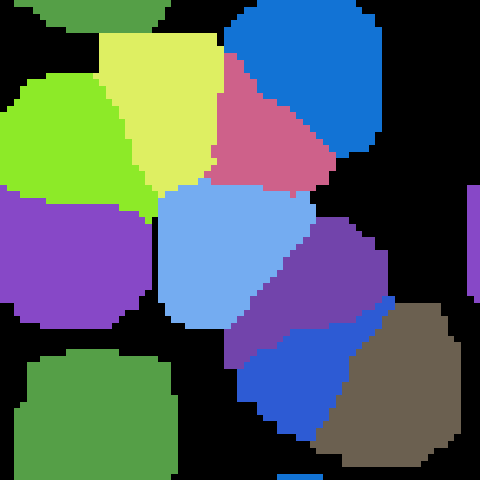
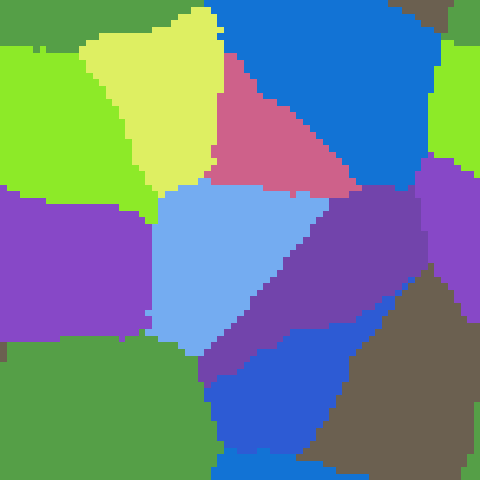
b.)

c.)

d.)

Rysunek 3.1 Rozrośnięta przestrzeń, przy zastosowaniu sąsiedztwa: a.) Moore’a,  
b.) von Neuman’a, c.) Heksagonalne (losowe), d.) pentagonalne.

Dobrowolnym wyborem, jest również rodzaj sąsiedztwa. Dobór sąsiedztwa ma wpływ na kształt otrzymanej przestrzeni po rozrośnięciu się wszystkich ziaren, jak to zostało uwidocznione na rysunku 3.1. Ciekawym rodzajem sąsiedztwa jest heksagonalne lewe i prawe, ponieważ są to dwa rodzaje sąsiedztw, które mogą zostać użyte jednocześnie, z zastosowaniem efektu probabilistycznego (Rysunek 3.2), ponieważ w każdym kroku czasowym losowana jest orientacja sąsiedztwa: lewe, prawe.



Rysunek 3.2 Rozrost ziaren przy sąsiedztwie heksagonalnym losowym.

Reguły określające algorytm rozrostu ziaren są proste i mówią, że jeżeli w otoczeniu rozpatrywanej komórki znajduje się komórka innego ziarna, a rozpatrywana komórka jeszcze nie przynależy do żadnego ziarna, to przyjmuje ona id sąsiada. Ponadto jeśli w otoczeniu rozpatrywanej komórki, znajduje się więcej komórek to, aktualna komórka przyjmuje id jej najliczniejszego sąsiada. A w przypadku, gdy liczba najliczniejszych sąsiadów jest większa niż jeden, to rozpatrywana komórka przyjmuje id losowo, jednego z jej najliczniejszych sąsiadów. Ten ostatni warunek wnosi koleją pro balistyczną cechę do algorytmu.

W ten sposób wygenerowana przestrzeń posiada dwie zmienne wewnętrzne, którymi są id komórek, które przypisują je do poszczególnych ziaren i informację o tym, czy komórka znajduje się na granicy ziaren. Granica ziaren odgrywa znaczącą rolę w procesach opisanych w kolejnych rozdziałach pracy.

## REKRYSTALZACJA DYNAMICZNA

### Opis zjawiska

Rekrystalizacja jest procesem metalurgicznym, mającym na celu przywrócenie zgniecionemu metalowi poprawnej struktury krystalicznej, jak również własności fizycznych oraz mechanicznych jakimi materiał charakteryzował się przed przeróbką. Dzieje się tak na skutek powstawania nowych zarodków (nie odkształconych ziaren, jak w przypadku zdrowienia), które następnie się rozrastają. Zarodki nowych ziaren powstają w miejscach o dużej koncentracji energii, np. na granicach ziaren, przecięciach bliźniaków, czy cząstkach obcych faz. Rozmiar ziarna jest zależny od szybkości tworzenia zarodków i ich wzrostu: czyli grube ziarna powstaną przy małej szybkości zarodkowani i dużej szybkości ich wzrostu. Rekrystalizacja jest procesem podobnym, lecz dużo bardziej efektywnym od zdrowienia. Polega na usuwaniu nadmiaru energii wprowadzonej do kryształu przez odkształcenie plastyczne. Duża ilość defektów struktury sieciowej oraz duża gęstość dyslokacji prowadzi do tego, że zgnieciony metal posiada większą energię wewnętrzną, niż metal wyżarzony lub odlany. Następuje uwolnienie zmagazynowanej energii, co jest spowodowane przede wszystkim zanikiem dyslokacji, których rozmiar spada o kilka rzędów wielkości. Skutkuje to zmniejszenie umocnienia, a co za tym idzie obniżenie twardości i wytrzymałości, natomiast następuje wzrost własności plastycznych materiału. Metal usiłując wyzwolić nadmiar energii, próbuje przejść ze stanu metastabilnego, do stanu równowagi termodynamicznej. Proces ten odbywa się bardzo szybko w wyższych temperaturach (powyżej tak zwanej temperatury rekrystalizacji), natomiast w temperaturze pokojowej jest on bardzo powolny. Dlatego, że temperatura ta zależy od wielu czynników takich jak: składu stopu, czasu wyżarzania, temperatury topnienia, czystości metalu, oraz wielkości zmagazynowanej energii odkształcenia, nie może być jednoznacznie określona. Powracanie metalu do stanu równowagi składa się z czterech etapów: zdrowienie, rekrystalizacja pierwotna, rozrost ziaren i rekrystalizacja wtórna.

### Implementacja modelu

Przystępując do implementacji modelu rekrystalizacji dynamicznej, na początku należy przeanalizować cztery podstawowe etapy konstruowania modelu:

* zdefiniowanie zmiennych stanu, poprzez określenie przestrzeni automatów
* wybór siatki komórek (zdefiniowanie otoczenia i warunków brzegowych)
* wybór sąsiedztwa
* opracowanie reguł przejścia

W pierwszym etapie musimy przygotować przestrzeń automatów, na której zajdzie proces rekrystalizacji. Przestrzenią nazywamy skończony zbór komórek, a każdą komórkę możemy przypisać do konkretnego ziarna. Przypisanie komórki do ziarna realizuje się dzięki zastosowaniu zmiennych dyskretnych. W jądrze rekrystalizacji ziarna reprezentowane są przez id, które można odczytywać i zapisywać za pomocą metod CELL\_GRAIN(gx, gy) i CELL\_GRAIN\_W(gx, gy). Taką przestrzeń można uzyskać np. dzięki naiwnemu rozrostowi ziaren. Poza id ziarna model rekrystalizacji dynamicznej posiada również, zmienne stanu o charakterze ciągłym, które reprezentują lokalną gęstość dyslokacji ρ i pełnią bardzo ważną rolę przy przejściu reguł. Dodatkowo został wprowadzony szereg innych zmiennych „przechowywanych w” ziarnach, co zostało narzucone przez specyficzną implementację jądra dla OpenCL i co zostanie opisane w następnym podrozdziale.

Kolejnym etapem jest zdefiniowanie otoczenia, czyli podanie promienia w obrębie jakiego będą rozpatrywani sąsiedzi. W zaimplementowanym modelu promień wynosi jeden, czyli analizowane są sąsiedzi leżący bezpośrednio obok rozpatrywanej komórki. Dodatkowo na tym etapie określamy warunki brzegowe. Do tego modelu zaimplementowano periodyczne warunki brzegowe na wszystkich ścianach przestrzeni. Periodyczne warunki brzegowe zostały opisane w podrozdziale 2.3.

Wybór sąsiedztwa w przypadku rekrystalizacji jest dowolny, ale należy pamiętać, że dla każdego sąsiedztwa rekrystalizacja będzie przebiegać inaczej. W przypadku zaimplementowanego modelu rekrystalizacji użyto sąsiedztwa von Neumana (Rysunek 3.3)

Rysunek 3.3 Sąsiedztwo von Neumana.

Ostatnią, a zarazem najważniejszą rzeczą, jaką należy określić to reguły przejścia. Są najważniejsze, ponieważ to one determinują działanie całego modelu. W przypadku zaimplementowanego modelu możemy podzielić reguły przejścia na dwie grupy. Pierwsza odpowiada za zarodkowanie nowych ziaren w przestrzeni automatów komórkowych. Druga reguła mówi jak ziarno propaguje, czyli rozrasta się w przestrzeni. Zaimplementowany model, nie jest wiernym odwzorowaniem wszystkich realnych zjawisk fizycznych, jakie zachodzą, podczas rekrystalizacji, ale ma na celu tylko odwzorowanie modelu, aby można było dokonać późniejszej analizy wydajności sprzętu sterowanego za pomocą architektury OpenCL.

if(isOnBoard == 1 &&

CELL\_REAL(gx, gy, 0) > criticalEnergy &&

CELL\_INT(gx, gy, 0) == 0 &&

recrystNghPrv == 0 &&

greaterEnergyThanNgh >= (NUM\_NEIGH - 1)){

if( ((double)(rand % 10) \* 0.1) <= 0.4){

if (rand % 2 == 0){

recrystallized = 1;

CELL\_GRAIN\_W(gx, gy) = newGrain;

GRAIN\_W(gx, gy) = GRAIN\_Int(0, 0, 2);

GRAIN\_Int\_W(0, 0, 2) =

GRAIN\_Int(0, 0, 2) + 1;

}

else {

CELL\_REAL\_W(gx, gy, 0) =

0.5 \* CELL\_REAL(gx, gy, 0);

}

}

Listing kodu 3.2 Reguła rekrystalizacji dynamicznej dotycząca zarodkowania.

Pierwsza reguła odnosząca się do zarodkowania nowego ziarna mówi, że gdy w komórce, która znajduje się na granicy ziarna isOnBoard == 1, zostanie przekroczona krytyczna wartość dyslokacji (kodzie zaimplementowane jest to jako: CELL\_REAL(gx, gy, 0) > criticalEnergy), komórka ta ulega rekrystalizacji recrystallized = 1, a jej wartość dyslokacji spada do wartości odniesienia. Pula dyslokacji jest obliczana w każdym kroku czasowym i rozdzielana jest ona do wszystkich komórek w przestrzeni:

- proces umocnienia (3.2)

- proces zdrowienia (3.3)

Jest to tak zwane działo dyslokacji, które komórkom leżącym na granicy przydziela większą wartość dyslokacji, określaną odpowiednim współczynnikiem prawdopodobieństwa, niż komórkom nie leżącym na granicy.

Ponadto algorytm musi sprawdzać inne warunki, dotyczące tego czy komórka już przypadkiem nie zrekrystalizowała CELL\_INT(gx, gy, 0) == 0, w jednym z poprzednich króków.

if (recrystNghPrv != 0 &&

greaterEnergyThanNgh >= (NUM\_NEIGH - 1) &&

CELL\_INT(gx, gy, 0) == 0){

int numGrain = rand % recrystNghPrv;

int iter = 0;

for(int i = 0; i< NUM\_NEIGH; i++){

temp = getNeighbourhood(gx, gy, i);

if(CELL\_INT(temp.x, temp.y, 0) == 1){

if(iter == numGrain){

recrystallized = 1;

CELL\_GRAIN\_W(gx, gy) =

CELL\_GRAIN(temp.x, temp.y);

CELL\_REAL\_W(gx, gy, 0) = 0;

}

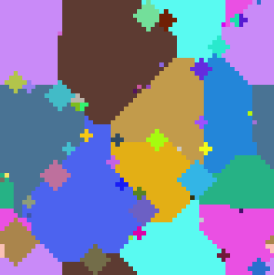
}

}

Listing kodu 3.3 Reguła rekrystalizacji dynamicznej dotycząca propagacji.

Druga reguła tycząca się propagacji, mówi że gdy w poprzednim kroku czasowym, któryś z sąsiadów rozpatrywanej komórki zrekrystalizował: recrystNghPrv != 0 i gęstość dyslokacji w rozpatrywanej komórce jest większa, niż w komórkach sąsiadujących: greaterEnergyThanNgh >= (NUM\_NEIGH - 1), to również i ta komórka ulega rekrystalizacji, a jej gęstość dyslokacji ustawiana jest na poziomie odniesienia. Oczywiście propagacja zachodzi, pod warunkiem, że rozpatrywana komórka jeszcze nie zrekrystalizowała w którymś z poprzednich kroków czasowych: CELL\_INT(gx, gy, 0) == 0.

Zaimplementowany w ten sposób model nie jest wiernym odwzorowaniem rekrystalizacji dynamicznej, ale daje możliwość odwzorowania wielkości ziaren w mikrostrukturze po zrekrystalizowaniu. Model został uproszczony, dzięki zastąpieniu równań określających kinematykę wzrostu ziaren, barierą energetyczną. Również została pominięta orientacja ziaren, która znacząco nie wpływa na strukturę końcową, ponieważ orientacja ta wykorzystana jest wyłącznie do określenia zderzenia ziaren. Wszystkie te uproszczenia jednak nie wpływają znacząco na jakość metody i bez ograniczeń może być to sposób, na generowanie reprezentacji początkowej mikrostruktury dla dalszych obliczeń.



a.)

b.)

c.)

d.)

Rysunek 3.4 . Rekrystalizacja dynamiczna, która została przeprowadzona na przestrzeni a.),   
b.) pojawienie się zarodów rekrystalizacji, c) rozrost zarodków, d.) zrekrystalizowana  
struktura.

### Model rekrystalizacji dynamicznej do zastosowań GPU

Podstawowym problemem przy implementacji modelu dynamicznej rekrystalizacji, w oparciu o jądro dla architektury, jest brak możliwości korzystania ze zmiennych globalnych. Są to ograniczenia płynące wprost z architektury OpenCL, która wprawdzie daje możliwość korzystania ze zmiennych globalnych, ale te zmienne muszą być typu stałego i muszą być zainicjalizowane przed wysłaniem ich do jądra. Nie ma natomiast możliwości korzystania ze zmiennych globalnych, które mogą być trwale zmodyfikowane w obrębie pracy całej grupy i w egzekucji jądra programu. Przykładem takiej zmiennej globalnej jest pula dyslokacji rozdzielenia pomiędzy komórki, wyliczona w każdym kroku iteracji wywoływania jądra, w obrębie którego przebiega (w danej jednostce czasowej) proces rekrystalizacji. Problem ten został rozwiązany poprzez wykorzystanie zmiennych przynależnych do każdej komórki. Architektura, na której uruchamiana jest aplikacja, daje możliwość zdefiniowania (przed wykonaniem jądra) zmiennych typu całkowitego i zmiennoprzecinkowego, przynależnych do danej komórki, lub ziarna. Takie zmienne są definiowane do tego, aby przechowywały w sobie numer iteracji wywołania jądra. Na podstawie iteracji przechowywanej w komórce, wyliczana jest wartość dyslokacji dla każdej komórki z osobna. Rozwiązanie to jest mało optymalne, ze względu na to, że operacja wyliczania dyslokacji dla ziarna jest kosztowna, ale konieczne.

Kolejnym utrudnieniem przy pisaniu kodu dla jądra programu, który jest zgodny ze standardem języka C99, jest mierzenie czasu. Jądro programu, które jest uruchamiane przez GPU, nie może odpytywać o takty procesora. W implementacji modelu utrudnienie to pojawia się przy konieczności liczenia wartości puli dyslokacji, które jest wyliczane ze wzoru (5.1) i jest zależne od czasu. Rozwiązanie tego problemu zostało uzyskane dzięki wykorzystaniu wartości kroku czasowego i przy sztywnym założeniu, że poszczególne iteracje są wykonywane, przy stałej różnicy Δt = 0.001. Czas wyliczony jest jako iloczyn różnicy czasowej i kroku czasowego, a następnie podstawiany do wzoru. Rozwiązanie to może zostać zastosowane, ponieważ implementacja modelu rekrystalizacji dynamicznej opiera się na dyskretnym modelu automatów komórkowych, dlatego też czas może być również mierzony ze stałą różnicą czasową.

Kolejnym utrudnieniem wynikającym z braku możliwości mierzenia czasu, jest brak dostępu do korzystania ze zmiennych losowych, wyliczanych w standardowy sposób, za pomocą CPU. Aby rozwiązać ten problem został wykorzystany mały (pod względem rozmiarowym) i szybki generator liczb losowych MWC64X RNG[[12]](#footnote-12). Propagacja wykorzystywana jest w regułach przejścia, które uwzględniają pewien współczynnik prawdopodobieństwa. Należy pamiętać, że generowanie zmiennych losowych jest kosztowne, dlatego też, w jądrze taka zmienna jest generowana tylko raz i wykorzystywana niezależnie w kilku miejscach modelu.

Liczby losowe w dynamicznej rekrystalizacji są potrzebne także do definiowania id nowych ziaren, które zarodkowały w wyniku przekroczenia dyslokacji krytycznej. O ile utrudnienie z liczbami losowymi jest możliwe do rozwiązania, to warunek aby te zmienne były unikatowe dla każdego ziarna jest już bardziej skomplikowany, ponieważ gdy przestrzeń jest dzielona pomiędzy różne rdzenie i gdy przestrzenie te są od siebie niezależne – nie ma możliwości korzystania ze zmiennych globalnych, które zawierałyby informacje o tym jakie id zostały już przydzielone do nowych ziaren i jakie są jeszcze dostępne. Problem ten musiał zostać rozwiązany w mało optymalny sposób, który zakłada, że id nowego ziarna równe jest indeksowi wektora, który powstał w wyniku przeniesienia macierzy, reprezentującej przestrzeń automatów komórkowych.

## PĘKANIE ZMĘCZENIOWE

### Opis zjawiska

W inżynierii materiałowej zmęczenie jest stopniowym i miejscowym strukturalnym uszkodzeniem materiału, który jest poddawany cyklicznym obciążeniom. 90 % zniszczonych podczas eksploatacji ruchomych części maszyn ma charakter zmęczeniowy, dlatego też badanie zmęczenia materiałów jest istotnym punktem inżynierii materiałowej. Warunkiem jest, aby wartość maksymalnych naprężeń była mniejsza od ostatecznego limitu na rozciąganie, ale może znajdować się ma granicy naprężenia plastycznego materiału. Zmęczenie pojawia się, gdy materiał jest poddawany obciążeniu i odciążeniu. Jeżeli te obciążenia są powyżej pewnego progu, na powierzchnie materiału pojawiają się mikroskopijne pęknięcia. Gdy te pęknięcia osiągną krytyczną wartość, następuję pękniecie całego materiału.

W przypadku stali, poniżej pewnej amplitudy naprężeń próbka nie ulegnie zniszczeniu – wytrzymałość zmęczeniowa w przypadku innych stopów amplituda naprężeń maleje ze wzrostem liczby cykli powodujących zniszczenie. Kształt struktury ma znaczący wpływ na trwałość zmęczeniową: otwory krawatowe lub ostre prowadzą do podwyższonych naprężeń lokalnych, w których mogą zostać zainicjowane pęknięcia zmęczeniowe. Otwory okrągłe i łagodne przejścia między komórkami są zatem ważne przy zwiększaniu wytrzymałościowej konstrukcji. Pękaniu zmęczeniowemu sprzyjają koncentratory – karby, ostre podcięcia itp. Zwiększenie wytrzymałości zmęczeniowej osiąga się przez eliminację lub ograniczenie miejsc zarodkowania pęknięć Inny sposób: wprowadzenie do warstwy powierzchniowej naprężeń ściskających lub jej umocnienie.

### Implementacja modelu

Na przestrzeni ostatnich lat wzrosło zainteresowanie komputerowymi modelami pękania, które posiadają cechy probabilistyczne, jak również deterministyczne. Podczas przeprowadzania badań uwaga jest skupiana na symulacji zjawisk mikrostrukturalnych, wykorzystujących metody dyskretne. Do metod dyskretnych należą automaty komórkowe, o które oparta jest niniejsza praca inżynierska.

Przy implementacji modelu pękania zmęczeniowego potrzebne jest zdefiniowanie dwóch reguł. Pierwsza z nich określa zarodkowanie, które w przypadku tego modelu jest rozumiane, jako powstanie ubytku w strukturze materiału. Ubytek taki nie jest ziarnem, dlatego nie przypisywane jest mu żadne id. Druga reguła określa propagację, czyli rozrastanie (powiększanie) się pęknięcia w strukturze materiału. Podobnie jak w przypadku rekrystalizacji dynamicznej, na początku modelowania zjawiska, należy zdefiniować przestrzeń automatów komórkowych. Przestrzeń ta jest określana, przed przeprowadzeniem symulacji i dostarczana do platformy. Może ona zostać wygenerowana, przy pomocy naiwnego rozrostu ziaren. W strukturze początkowej na której będzie zachodziło zjawisko, nie ma pęknięć, w postaci ubytków komórek, które nie posiadają jeszcze przydzielonego id. Każda komórka prócz id, posiada informację czy jest ona aktywna, to znaczy – czy jest ona interpretowana, jako ubytek w strukturze, lub informację o tym, czy jest ona nieaktywna i ma być interpretowana, jako ośrodek stały. Ponadto komórka musi posiadać rzeczywistą zmienną wewnętrzną, która określa krytyczną wartość naprężenia komórki i jest w stanie spowodować zarodkowanie pęknięcia w danej komórce. Drugą zmienną wewnętrzną jest lokalne naprężenie, występujące w rozpatrywanej komórce i które jest porównywane z naprężeniem krytycznym. Po za zmiennymi wewnętrznymi określona jest zmienna zewnętrzna, która nie jest wymagana przy określaniu przestrzeni, ponieważ pochodzi z zewnątrz. Zmienna ta to naprężenie jakie działa na strukturę.

Dla tak zdefiniowanej przestrzeni, należy określić również warunki brzegowe, które w tym modelu są niestandardowe, z tego względu, że łączą w sobie warunki periodyczne i zamknięte pochłaniające. Warunki periodyczne zostały opisane, w podrozdziale „Automaty komórkowe” i przy okazji omawiania procesu rekrystalizacji dynamicznej w niniejszej pracy. Natomiast warunki pochłaniające to takie, w których siatka jest zdefiniowana w ten sposób, że jej brzegi są wypełnione z góry ustaloną wartością. Te dodatkowe wartości na brzegach siatki, również ma wpływ na zachowanie automatu, poprzez funkcję przejścia. Zastosowanie różnych warunków brzegowych podyktowane jest otoczeniem w jakim znajduje się materiał poddawany obciążeniom. Jak jest to zaznaczone na rysunku 3.5, który przedstawia kwadratową przestrzeń automatów komórkowych z dwoma ziarnami o id 1 i 3. Na ścianach pionowych zdefiniowane są warunki brzegowe, ponieważ tam działa zewnętrzna siła rozciągająca próbkę, działająca w kierunku poziomym. Natomiast na ścianach poziomych przestrzeni zastosowane zostały warunki zamknięte pochłaniające, ponieważ na brzegu dolnych i górnym przykładowej struktury następuje jest zetknięcie z przestrzenią otaczającą. Dlatego też w przypadku rozpatrywania przestrzeni, która odwzorowuje strukturę danego materiału, należy jeszcze uwzględnić dodatkowe komórki, którymi jest ona otoczona. Ilość tych komórek jest zależna od ustalonego promienia sąsiedztwa.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 |
| 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 |
| 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 |
| 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 |
| 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |

**ρ**

**ρ**

Ośrodek zewnętrzny

Ośrodek zewnętrzny

Rysunek 3.5 Przestrzeń dla pękania zmęczeniowego z warunkami brzegowymi periodycznymi i zamkniętymi pochłaniającymi.

Po zdefiniowaniu przestrzeni należy określić kąt α nachylenia granic, w stosunku do siły działającej na strukturę. Taki kąt α jest potrzebny do wyznaczenia naprężeń lokalnych, działających na daną komórkę. Za pomocą tego kąta wyznaczany jest wektor normalny *n* (prostopadły) jednostkowy do granicy ziarna. Wektor normalny wyznacz się według wzoru (3.4).

*n* = [cos(α), sin(α)] (3.4)

W takim modelu można przetestować różne sąsiedztwa. W niniejszej pracy zostało zastosowane sąsiedztwo von Neumanna. Dla tak zdefiniowanej przestrzeni automatów komórkowych należy określić tylko jeszcze reguły przejścia. Funkcje przejścia zależą od stanu komórki która jest rozpatrywana, a także od komórek sąsiadujących z rozpatrywaną komórką.

if(CELL\_INT(gx, gy, 0) == 0 &&

activeNghNo == -1 &&

CELL\_REAL(gx, gy, 0) != 0 &&

sigma\_local > sigma\_cr &&

P < L){

CELL\_GRAIN\_W(gx, gy) = 1;

}

else{

if(CELL\_INT(gx, gy, 0) == 0 &&

activeNghNo != -1 &&

((CELL\_REAL(gx, gy, 0) !=0 &&

P < L) ||

P < ZetaL) &&

sigma\_local > XiSigma\_cr){

CELL\_GRAIN\_W(gx, gy) = 1;

}

}

Listing kodu 3.4 Reguła przejścia dla pękania zmęczeniowego.

Tak jak już to zostało wspomniane, pierwsza reguła decyduje o zarodkowaniu, czyli powstawaniu nowych ubytków w strukturze materiału. Pierwszym warunkiem jaki musi zostać spełnionym to, aby rozpatrywana komórka nie była aktywna CELL\_INT(gx, gy, 0) == 0, czyli w rozpatrywanej komórce nie było ubytku. Aby mogło zajść zarodkowanie, również komórki sąsiadujące z rozpatrywaną nie mogą być aktywne: activeNghNo == -1. Warunkiem koniecznym, ale nie wystarczającym jest aby komórka, która potencjalnie ma zarodkować znajdowała się na granicy ziaren: CELL\_REAL(gx, gy, 0) != 0. Warunek ten wynika z tego, że komórki należące do różnych ziaren, mają mniejsze oddziaływanie na siebie, niż komórki pochodzące z tego samego ziarna. Jeżeli już poprzednie warunki zostaną spełnione, sprawdzane jest czy lokalna krytyczna wartość naprężenia, jest większa od naprężenia działającego z zewnątrz na całą strukturę: sigma\_local > sigma\_cr. Naprężenie lokalne krytyczne wyliczane jest z wykorzystaniem wektora normalnego do granicy ziarna. Nachylenie ziarna ma duży wpływ na zdolność komórki do zarodkowania, ze względu na siły wypadkowe pochodzące od sąsiadów, rozkładające się w tym ziarnie. I tak, gdy wektor normalny jest równoległo do działającego naprężenia zewnętrznego, ziarno przejawia największą zdolność do pękania, a gdy ziarno znajduje się na granicy, której wektor normalny jest prostopadły do danego naprężenia, to prawdopodobieństwo nastąpienia zarodkowania, jest zbliżone do zera. Jeżeli wszystkie powyższe warunki sprzyjające zarodkowaniu zostaną spełnione, nie oznacza to, że ziarno zmienia swój stan na aktywny, ponieważ zjawisku pękania towarzyszy jeszcze pewien współczynnik prawdopodobieństwa: P < L.

Jeżeli w strukturze pojawiły się jakieś zarodki pękania, to może wtedy zachodzić propagacja sąsiadów komórek aktywnych. Jest to pierwszy warunek, konieczny do zajścia propagacji – sąsiad rozpatrywanej komórki musiał zrekrystalizować w poprzednim kroku czasowym: activeNghNo != -1, a rozpatrywana komórka nie może być aktywna w aktualnym kroku czasowym: CELL\_INT(gx, gy, 0) == 0. Kolejny warunek mówiący o tym, że komórka pęknie z pewnym prawdopodobieństwem, jest zależny od tego czy rozpatrywana komórka, znajduje się na granicy ziaren, czy wewnątrz swojego ziarna. Jeżeli komórka, znajduje się na granicy ziarna, to prawdopodobieństwo jej propagacji jest dużo większe, aniżeli by to miało miejsce w przeciwnym przypadku: ((CELL\_REAL(gx, gy, 0) !=0 && P < L) || P < ZetaL). Ostatnim warunkiem propagacji jaki musi zostać spełniony, jest to aby lokalna wartość naprężenia była większa, od krytycznej wartości naprężenia, jakie może działać na komórkę: sigma\_local > XiSigma\_cr. Różnicą w tym warunku w porównaniu do analogicznego warunku, z reguły dotyczącej zarodkowania jest to, że wartość krytyczna naprężenia jest pomnożona, przez współczynnik, który zmniejsza siłę tego naprężenia krytycznego.



a.)

l.)

b.)

m.)

n.)

k.)

j.)

o.)

e.)

g.)

d.)

c.)

i.)

h.)

f.)

Rysunek 3.6 Pękanie zmęczeniowe na granicy dwóch ziaren.

Rysunek 3.6 przedstawia przykład pękania zmęczeniowego, na granicy dwóch ziaren 3.6a (fragment przestrzeni pokazany w czasie gdy pękanie jeszcze nie nastąpiło), przy czym zaprezentowany fragment przestrzeni przestawia granicę z dwoma kątami nachylenia w odniesieniu do siebie. Na kolejnych rysunkach zaprezentowane jest pękanie materiału w miarę upływu czasu. Na rysunkach 3.6b i 3.6l widać proces zarodkowania, na pozostałych rysunkach natomiast w większości przypadków można zauważyć zjawisko propagacji.

### Model pękania zmęczeniowego do zastosowań GPU

W implementacji jądra powyższego modelu dostosowanego do architektury OpanCL, konieczne jest korzystanie z liczb losowych. Generowanie liczb losowych we fragmencie programu wykonywanego na procesorach kart graficznych, jak już wspomniano w niniejszej pracy jest problematyczne, ze względu na to wymaga dużych zasobów obliczeniowych.

W powyższym modelu zostało zastosowane uproszczenie, ponieważ w przypadku gdy w rzeczywistym materiale następuje pęknięcie materiału (pojawiają się zarodki pękania), cała struktura zmienia swoją objętość. W niniejszej implementacji, nie uwzględniono zmiany objętości materiału po nastąpieniu pęknięcia, co nie ma znaczącego wpływu na otrzymany wynik, po zajściu procesu pękania zmęczeniowego. W rzeczywistości w miejscu, w którym pojawia się pęknięcie, powinna pojawić się tylko pusta przestrzeń, gdzie ono nastąpiło. Tak dokładna symulacja zjawiska, jest bardzo trudna do zaimplementowania w jądrze, którego kod wynikowy wysyłany jest do GPU i CPU i wykraczałaby znacząco po za ramy tematyczne niniejszej pracy. Uproszczenie, które zostało zastosowane, polega na rozpatrywaniu komórki, a nie przestrzeni między dwoma komórkami, a w przypadku spełnienia jednej z dwóch reguł przejścia, zmieniany jest stan danej komórki na aktywny, czyli komórka traktowana jest jako szczelina po pęknięciu.

# WYNIKI

## Testy wydajnościowe zaimplementowanych modeli mikrostruktur materiałowych, przy różnej ilość procesorów GPU

Celem niniejszej pracy nie jest testowanie wydajności przedstawionych i zaimplementowanych algorytmów, dlatego obliczenia dla niniejszych testów zostały przeprowadzone na modelu pękania zmęczeniowego. Dodatkowym argumentem uzasadniającym wykonanie obliczeń tylko na jednym z modeli, są dane zebrane w tabeli 4.1. Czasy uruchamiania poszczególnych modeli, przy zastosowaniu różnych konfiguracji zestawienia platform, na których zostały one uruchomione, pokazują że złożoności obliczeniowe naiwnego rozrostu ziaren, rekrystalizacji dynamicznej i pękania zmęczeniowego, są bardzo podobne do siebie. Jest tak, ponieważ implementacja modeli mikrostruktur materiałowych w oparciu o metodologię GPGPU jest schematyczna. Modele mikrostruktur materiałowych implementowane zgodnie z ideą zaprezentowaną w niniejszej pracy, różnią się między sobą przede wszystkim regułami przejścia je determinującymi. Reguły przejścia natomiast są implementowane przede wszystkim za pomocą instrukcji warunkowych, które mają znacznie mniejszą złożoność obliczeniową, jaką mają na przykład instrukcje sterujące. Poniższe obliczenia wykonano na karcie graficznej Nvidia Tesla M2090.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **1 x GPU** | | **Czas (s)** | | |
| **Wymiar przestrzeni** | **Ilość iteracji** | Naiwny rozrost | Rekrystalizacja dynamiczna | Pękanie zmęczeniowe |
| **100x100** | **100** | 4.55 | 4.83 | 4.76 |
| **200x200** | **200** | 37.98 | 38.11 | 37.88 |
| **500x500** | **500** | 223.45 | 247.70 | 237.47 |
| **1000x1000** | **1000** | 1798.02 | 1894.04 | 1827.13 |

Tabela 4.1 Zestawienie czasu obliczeń trzech zaimplementowanych modeli mikrostruktur materiałowych.

Poniżej przedstawiono czasy wykonywania modelu pękania zmęczeniowego na różnych platformach. Aplikacja została uruchomiona na jednym procesorze CPU, jednym, dwóch i czterech procesorach GPU, karty graficznej NVidia Tesla M2090. Dodatkowo na każdej platformie testowano różne warianty parametrów wejściowych aplikacji, zmieniając wymiar przestrzeni na: 32x32, 100x100, 200x200, 500x500 i 1000x1000, komórek w orientacji pionowej i poziomej. Każdą przestrzeń na każdej z platform podawano obliczeniom w różnej ilości kroków: 100, 200, 500 i 1000. Przedstawione poniżej czasy to średnia z trzech pomiarów.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **1 x GPU** | | | | | |
| **Czas (s)** | | Ilość iteracji | | | |
| **100** | **200** | **500** | **1000** |
| Wymiar przestrzeni | **32x32** | 1.33 | 2.73 | 2.50 | 13.50 |
| **100x100** | 4.76 | 9.50 | 10.75 | 48.33 |
| **200x200** | 18.74 | 37.88 | 41.97 | 191.36 |
| **500x500** | 104.84 | 212.98 | 237.47 | 1071.01 |
| **1000x1000** | 437.85 | 900.93 | 1022.00 | 1827.34 |

Tabela 4.2 Zestawienie czasu obliczeń dla różnych przestrzeni, przy różnej ilości iteracji na jednym procesorze graficznym GPU.

Wykres 4.1 Zestawienie czasu obliczeń w sekundach dla różnych przestrzeni, przy różnej ilości iteracji na jednym procesorze graficznym GPU.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **2 x GPU** | | | | | |
| **Czas (s)** | | Ilość iteracji | | | |
| **100** | **200** | **500** | **1000** |
| Wymiar przestrzeni | **32x32** | 1.60 | 3.28 | 4.12 | 17.18 |
| **100x100** | 4.89 | 9.90 | 12.20 | 49.05 |
| **200x200** | 18.14 | 36.91 | 42.64 | 182.57 |
| **500x500** | 101.00 | 201.05 | 219.00 | 1006.69 |
| **1000x1000** | 427.15 | 861.33 | 994.72 | 3605:10.12 |

Tabela 4.3 Zestawienie czasu obliczeń dla różnych przestrzeni, przy różnej ilości iteracji na dwóch procesorach graficznych GPU.

Wykres 4.2 Zestawienie czasu obliczeń w sekundach dla różnych przestrzeni, przy różnej ilości iteracji na dwóch procesorach graficznych GPU.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **4 x GPU** | | | | | |
| **Czas (s)** | | Ilość iteracji | | | |
| **100** | **200** | **500** | **1000** |
| Wymiar przestrzeni | **32x32** | 2.32 | 4.42 | 5.01 | 22.83 |
| **100x100** | 5.36 | 10.67 | 11.90 | 53.64 |
| **200x200** | 18.12 | 36.86 | 40.63 | 181.71 |
| **500x500** | 97.23 | 193.90 | 203.14 | 962.26 |
| **1000x1000** | 406.16 | 811.01 | 954.32 | 3692.23 |

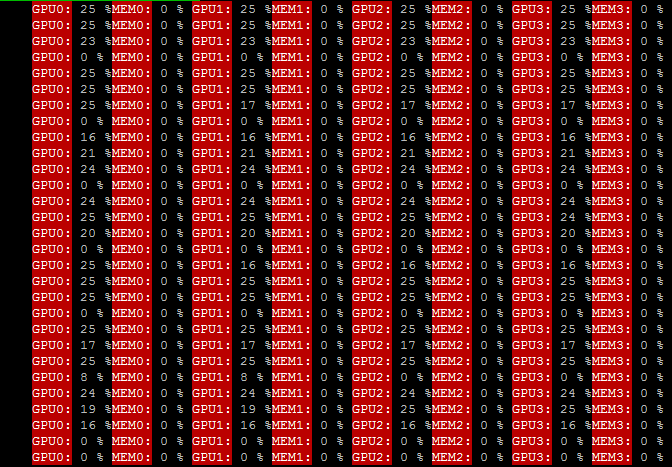
Tabela 4.4 Zestawienie czasu obliczeń dla różnych przestrzeni, przy różnej ilości iteracji na czterech procesorach graficznych GPU.

Wykres 4.3 Zestawienie czasu obliczeń w sekundach dla różnych przestrzeni, przy różnej ilości iteracji na czterech procesorach graficznych GPU.

Analizując powyższe wyniki, na podstawie tabel 4.2-4 i wykresów 6.1-3 dla poszczególnych platform, można zauważyć, że czas obliczeń dla małej przestrzeni o wysokości i szerokości 32-200 komórek, dla akceleratorów GPU maleje wraz ze zmniejszającą się liczbą klastrów na których te obliczenia są wykonywane. Jest to spowodowane kosztownym czasem komunikacji pomiędzy tymi urządzeniami. Wyniki te pokazują, że nie warto jest stosować GPGPU, dla małych obliczeń, ponieważ obliczenia nie są wystarczająco wydajne. Sytuacja zmienia się diametralnie w przypadku zwiększania się ilości obliczeń, związanych z rozmiarem przestrzeni i ilości operacji, którym podawane są poddawane te przestrzenie, co można zauważyć za pomocą wykresu zbiorczego 4.4 wszystkich platform.

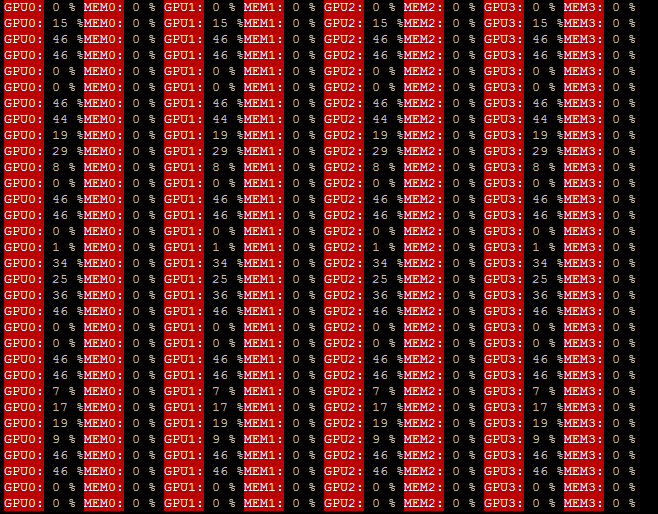
Wykres 4.4 Zestawienie czasu obliczeń (w sekundach) dla różnych przestrzeni, przy różnej ilości iteracji na różnej ilości procesorów graficznych GPU.

Lepszej analizie otrzymanych wyników może posłużyć rysunek X.X, który jest zrzutem ekranowym terminala, w którym został wywołany skrypt, do odpytywania procesorów graficznych z interwałem czasowym co 1 s.



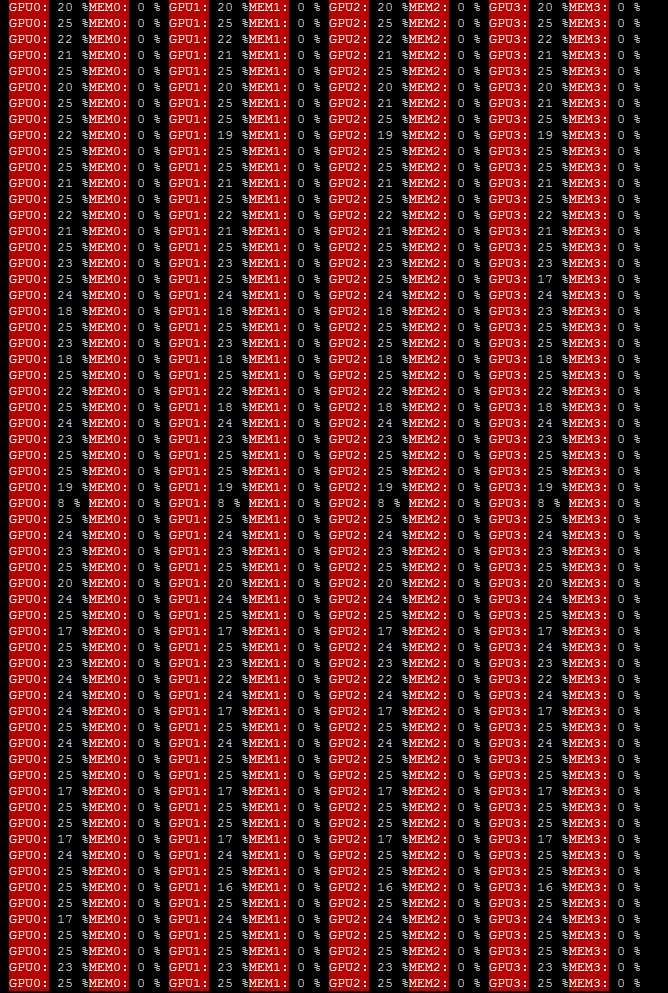
Rysunek 4.1 Zużycie akceleratorów czterech graficznych GPU, odczytywane co 1 s, dla modelu pękania zmęczeniowego uruchomionego na przestrzeni 200x200, przy 200 iteracjach.

Na załączonym rysunku 4.1 łatwo zauważyć, ze zużycie każdego z czterech uruchomionych procesorów graficznych GPU, oscyluje w granicy do 25% i co trzecie odpytanie wskazuje 0% zużycie. Zużycie 25% procesorów jest spowodowane małą przestrzenią, dla której została uruchomiona aplikacja: 200 na 200 komórek, ponieważ, dla tak małej przestrzeni procesory graficzne są w stanie skończyć swoją pracę jeszcze przed synchronizacją. Synchronizacja, jest widoczna na rysunku 4.1, w co trzecim wierszu, w którym to zużycie procesora jest równe 0%. To właśnie w tym momencie, dochodzi do największych strat, wspomnianych przy omawianiu czasu obliczeń, dla małych przestrzeni.



Rysunek 4.2 Zużycie akceleratorów czterech graficznych GPU, odczytywane co 1 s, dla modelu pękania zmęczeniowego uruchomionego na przestrzeni 1000x1000, przy 200 iteracjach.

Dla porównania rysunek 4.2 zawiera procentowe wykorzystanie akceleratorów GPU, dla tego samego modelu, tej samej ilości iteracji, ale przestrzeni powiększonej z 200 na 200 komórek do 1000 na 1000 komórek w przestrzeni. Analizując rysunki 4.1 i 4.2 zauważalne jest to, że przy większym rozmiarze przestrzeni automatów komórkowych wydajność procesorów wzrasta i również czas sekwencyjnej synchronizacji danych się wydłuża, z ok. trzech sekund do czterech sekund, po których wykorzystanie GPU spada do 0%.



Rysunek 4.4 Zużycie akceleratorów czterech graficznych GPU, odczytywane co 1 s, dla modelu pękania zmęczeniowego uruchomionego na przestrzeni 200x200, przy 1000 iteracjach.

Rysunek 4.4 to natomiast uruchomienie aplikacji, dla przestrzeni 200 na 200 komórek, na czterech akceleratorach graficznych GPU, ale z tą różnicą do uruchomienia z rysunku 4.1, że została zwiększona ilość operacji na poszczególnych przestrzeniach z 200 do 1000. Na załączonym rysunku 4.4 od razu widać, że interwały synchronizacji następuje co bardzo długie odstępy czasowe, a wykorzystanie akceleratorów spada tylko do 8 %.

Analiza ta pozwala stwierdzić, że zastosowanie obliczeń na akceleratorach GPU w oparciu o automaty komórkowe, jest bardziej efektywne dla większych złożoności obliczeń, większej przestrzeni danych i większej ilości operacji na przestrzeni.

## Porównanie obliczeń na CPU i GPU

Do obliczeń wykorzystano procesor sześciordzeniowy procesor Intel@ Xeon@ X5650 [[13]](#footnote-13)o częstotliwości taktowania 2.66 Ghz.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **1 x CPU** | | | | | |
| **Czas (s)** | | Ilość iteracji | | | |
| **100** | **200** | **500** | **1000** |
| Wymiar przestrzeni | **32x32** | 0.49 | 0.92 | 2.41 | 4.76 |
| **100x100** | 2.14 | 4.39 | 11.09 | 21.77 |
| **200x200** | 8.32 | 16.54 | 41.79 | 83.68 |
| **500x500** | 48.03 | 95.82 | 238.62 | 478.08 |
| **1000x1000** | 93.85 | 397.10 | 1041.32 | 1980.80 |

Tabela 4.5 Zestawienie czasu obliczeń w sekundach dla różnych przestrzeni, przy różnej ilości iteracji na jednym procesorze CPU.

Analizując również czasy wykonywania aplikacji, na procesorach CPU, dla tego samego zestawu danych co dla GPU, łatwo zauważyć, na podstawie tabeli 4.5 że odpowiednie czasy dla dowolnej liczby procesorów akceleratorów GPU (jednego, dwóch i czterech), czasy na CPU są znacząco krótsze. Należy zaznaczyć, że moc obliczeniowa procesora Intel@ Xeon@ X5650, o częstotliwości taktowania 2.66 Ghz, i pamięci podręcznej Cache jest w stanie przewyższyć moc obliczeniową kart Ncidia Tesla M2090, co przekłada się na powyższe wyniki. Na dłuższy czas wykonania obliczeń, na akceleratorach GPU wpływa również nie wykorzystanie w kodzie źródeł zaimplementowanych modeli, specyficznych zmiennych lokalnych wprowadzonych przez CUDA, do których dostęp jest bardzo szybki. Innym czynnikiem, wpływającą na opóźnienia obliczeń na GPU, może mieć wpływ tworzenie nowych wątków, na akceleratorach GPU, które radzą sobie z tym wolniej niż procesory CPU.

## Przyspieszenie obliczeń na procesorach graficznych

Jak to zostało opisane w podrozdziale „Przyspieszenie obliczeń równoległych”, przyspieszenie możemy policzyć jako iloczyn czasu rozwiązania zadania najlepszym algorytmem sekwencyjnym na pojedynczym procesorze, do czasu rozwiązania zadania rozważanym algorytmem równoległym na p procesorach. Wzór ten został opracowany dla zwykłych CPU, lecz jeżeli pojedynczy wieloprocesor strumieniowy akceleratora graficznego GPU, potraktowałoby się jako pojedynczy procesor, można pokusić się o próbę wyliczenia przyspieszenia dla procesorów graficznych.

W celu pomiarów przyspieszenia wykorzystanej do implementacji modeli materiałowych karty graficznej Nvidia Tesla M2090, wykorzystano jeden procesor graficzny GPU i na jego podstawie zmierzono przyspieszenie, jakie można uzyskać za pomocą odpowiedniej liczy jego procesorów strumieniowych. Do pomiarów wykorzystano modele: pękanie zmęczeniowe, rekrystalizacja dynamiczna i naiwny rozrost ziaren, wszystkie modele poddano obliczeniom, przy 100 iteracjach. W celu zmierzenia przyspieszenia, jako sekwencyjny pomiar używa się obliczenia na wybranym modelu z wykorzystaniem tylko jednego wieloprocesora strumieniowego. Liczbę wieloprocesorów strumieniowych można ustawić za pomocą funkcji enqueueNDRangeKernel pochodzącej z biblioteki OpenCl, która jako argumenty przyjmuje liczne wątków, jaką może być maksymalnie wykorzystana przez przestrzeń lokalną i globalna. Przykładowo aby obliczenia wykonać tylko na jednym wątku, należy przydzielić przestrzeni globalnej i lokalnej tylko jeden wątek, który może zostać obsłużony przez tylko jeden procesor strumieniowy. Ustawienie to polega na zmianie wartości liczby będącej argumentem funkcji NDRange, której wynik następnie przekazywany jest jako argument funkcji enqueueNDRangeKernel:

err = (\*queue).enqueueNDRangeKernel(

\*kernel,

cl::NullRange,

cl::NDRange(128),

cl::NDRange(16),

0,

&device->ev\_);

Listing kodu 4.1 Fragment kodu platformy, która jest odpowiedzialna za przydzielania zadań procesorom.

Poniższa tabele 4.6-8 przedstawiają czasy obliczenia przestrzenia dla reguł determinowanych przez wybrane modele mikrostruktury materiałowej, dla 100 iteracji. Ponadto w ostatniej kolumnie wyliczono przyspieszenia dla poszczególnej ilości procesorów strumieniowych.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pękanie zmęczeniowe | | | | | |
| Liczba komórek w przestrzeni | Liczba wątków przeznaczona na przestrzeń lokalną | Liczba wątków przeznaczona na przestrzeń globalną | Ilość wykorzystanych wieloprocesorów strumieniowych | Czas obliczeń (s) | Przyspieszenie |
| 250000 | 512 | 512 | 1 | 211.88 | 1.00 |
| 512 | 1024 | 2 | 109.11 | 1.94 |
| 512 | 2048 | 4 | 58.06 | 3.64 |
| 512 | 4096 | 8 | 32.95 | 6.39 |
| 512 | 8192 | 16 | 30.05 | 7.03 |

Tabela 4.6 Czasy obliczeń i przyspieszenie dla różnej ilości wieloprocesorów strumieniowych, model pękania zmęczeniowego.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Rekrystalizacja dynamiczna | | | | | |
| Liczba komórek w przestrzeni | Liczba wątków przeznaczona na przestrzeń lokalną | Liczba wątków przeznaczona na przestrzeń globalną | Ilość wykorzystanych wieloprocesorów strumieniowych | Czas obliczeń (s) | Przyspieszenie |
| 250000 | 512 | 512 | 1 | 225.36 | 1.00 |
| 512 | 1024 | 2 | 117.23 | 1.92 |
| 512 | 2048 | 4 | 61.71 | 3.69 |
| 512 | 4096 | 8 | 35.58 | 6.43 |
| 512 | 8192 | 16 | 32.61 | 7.03 |

Tabela 4.7 Czasy obliczeń i przyspieszenie dla różnej ilości wieloprocesorów strumieniowych, model rekrystalizacji dynamicznej.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Naiwny rozrost ziaren | | | | | |
| Liczba komórek w przestrzeni | Liczba wątków przeznaczona na przestrzeń lokalną | Liczba wątków przeznaczona na przestrzeń globalną | Ilość wykorzystanych wieloprocesorów strumieniowych | Czas obliczeń (s) | Przyspieszenie |
| 250000 | 512 | 512 | 1 | 205.74 | 1.03 |
| 512 | 1024 | 2 | 100.56 | 2.11 |
| 512 | 2048 | 4 | 56.32 | 3.77 |
| 512 | 4096 | 8 | 30.11 | 7.03 |
| 512 | 8192 | 16 | 28.95 | 7.54 |

Tabela 4.8 Czasy obliczeń i przyspieszenie dla różnej ilości wieloprocesorów strumieniowych, model naiwnego rozrostu ziaren.

Analiza otrzymanych przyspieszeń prowadzi do wniosków, że przyspieszenie obliczeń wzrasta wraz z liczba wykorzystywanych wieloprocesorów strumieniowych. Karta graficzna Nvidia Tesla M2090, dysponuje 32 rdzeniami (procesorami strumieniowymi SP), zgrupowanymi w 16 bloków zwanymi wieloprocesorami strumieniowymi SMs, w sumie jedna karta graficzna posiada 512 rdzeni. W powyższych obliczeniach przyjęto, że karta graficzna będzie wykonywała obliczenia sekwencyjne w momencie, gdy algorytm będzie wykorzystywał do obliczeń tylko jeden procesor wielordzeniowy składający się z 32 rdzeni. W tym celu za pomocą interfejsu biblioteki OpenCL, którego fragment został przedstawiony na listingu 4.1, przydzielono wątki przestrzeni lokalnej i globalnej automatów komórkowych. Aby w pełni wykorzystać moc obliczeniową wszystkich rdzeni wieloprocesowa strumieniowego, liczba wątków które zostały na nim uruchomione kilkudziesięciokrotnie przekracza liczbę rdzeni, które mogą je obsłużyć. Dzięki temu współbieżność, którą obsługują użyte procesory graficzne, będzie mogła zostać w pełni wykorzystana. Współbieżność polega na tym, że wątki uruchomione na tym samym procesorze są przełączane w krótkich przedziałach czasu, co sprawia wrażenie, że wykonują się równolegle. Efekty przyspieszenia, zwiększając liczbę wieloprocesorów strumieniowych można zaobserwować na poniższych wykresach:

Wykres 4.5 Przyspieszenie w zależności od ilości wieloprocesorów strumieniowych, dla modelu pękania zmęczeniowego.

Wykres 4.6 Przyspieszenie w zależności od ilości wieloprocesorów strumieniowych, dla modelu rekrystalizacji dynamicznej.

Wykres 4.7 Przyspieszenie w zależności od ilości wieloprocesorów strumieniowych, dla modelu naiwnego rozrostu ziaren.

Na podstawie analizy wykresów 4.5-7 można zaobserwować jak wzrasta przyspieszenie w zależności od ilości procesorów strumieniowych. Większa ilość użytych procesorów strumieniowych pomogła kilkukrotnym przyspieszeniu obliczeń. Szesnastokrotne przyspieszenie (zastosowanie szesnastu procesorów wielordzeniowych w stosunku, do obliczeń sekwencyjnych), pozwoliła na około siedmiokrotne przyspieszenie obliczeń. Na powyższych wykresach można również zauważyć, że przyspieszenie rzeczywiste zmierzone na trzech zaimplementowanych modelach mikrostruktur materiałowych, wraz ze zwiększaniem ilości wieloprocesorów strumieniowych, coraz bardziej odbiega od idealnego przyspieszenia obliczeń równoległych (które jest funkcją liniową).

## Skalowalność obliczeń na akceleratorach GPGPU

Przyspieszenie skalowalne to przyspieszenie, jakie uzyskuje się rozważając dla danej liczby procesorów p zadanie o rozmiarze p-krotnie większym od zadania rozwiązywanego na pojedynczym procesorze. W poprzednim podrozdziale zostało policzone przyspieszenie, które różniło się od przyspieszenia skalowalnego tym, że przestrzeń na której zostały przeprowadzone obliczenia nie zmieniała się wraz ze zwiększaniem liczby wieloprocesorów strumieniowych. W przypadku przyspieszenia skalowalnego, przestrzeń będzie zwiększała się z taką samą krotnością, jaki liczba wieloprocesorów strumieniowych, w stosunku ilości wieloprocesorów strumieniowych wykorzystanych do obliczeń sekwencyjnych, (czyli jednego wieloprocesora strumieniowego).



Tabela 4.9 Czasy obliczeń i skalowalność dla różnej ilości wieloprocesorów strumieniowych, model pękania zmęczeniowego.

Wykres 4.8 Skalowalność w zależności od ilości wieloprocesorów strumieniowych, dla modelu naiwnego rozrostu ziaren.



Tabela 4.10 Czasy obliczeń i skalowalność dla różnej ilości wieloprocesorów strumieniowych, model rekrystalizacji dynamicznej.

Wykres 4.9 Skalowalność dla modelu naiwnego rozrostu ziaren w zależności od ilości wieloprocesorów strumieniowych.



Tabela 4.11 Czasy obliczeń i skalowalność dla różnej ilości wieloprocesorów strumieniowych, model naiwnego rozrostu ziaren.

Wykres 4.10 Skalowalność dla modelu naiwnego rozrostu ziaren w zależności od ilości wieloprocesorów strumieniowych.

Analizując tabelę 4.9-11 z czasami dla poszczególnej liczby wieloprocesorów strumieniowych i rozmiar przestrzeni, oraz wyliczoną skalowalność przyspieszenia można zauważyć, że skalowalność dla wszystkich trzech modeli mikrostruktur materiałowych, na których zostały wykonane obliczenia jest do siebie bardzo zbliżony, jak to miało miejsce w przypadku przyspieszenia.

Liniowe przyspieszenie skalowalne uzyskuje się wtedy kiedy czas rozwiązania nie zmienia się przy rozwiązaniu zadania o rozmiarze p-krotnie większym, na p razy więcej procesorach – efektywność zrównoleglenia jest wtedy stała. Na wykresach 4.8-10 można porównać idealną skalowalność przyspieszenia z wyliczoną skalowalnością, dla poszczególnych modeli. Analizując powyższe wykresy można dostrzec podobnie, jak w przypadku przyspieszenia, że wyliczona skalowalność odbiega od ideału, coraz bardziej, wraz ze wzrostem liczby procesorów strumieniowych.

# WIOSKI

Celem niniejszej pracy było opracowanie i zaimplementowanie algorytmów, dzięki którym udowodniono zasadność stosowania obliczeń równoległych na akceleratorach GPU. Zaimplementowane modele nie są wiernym odwzorowaniem reprezentowanych przez nie zjawisk fizycznych. Mimo to wyniki obliczeń uzyskane dzięki nim są zbliżone do eksperymentalnych. Przy implementacji modeli mikrostruktur materiałowych, których środowiskiem uruchomieniowym mogą być akceleratory GPU, należy liczyć się z szeregiem utrudnień, których przezwyciężenie wynagrodzone jest dużą wydajnością kodu. Trzeba uwzględnić ograniczenia płynących z jeszcze niedostatecznego wsparcia języków programowania wysokiego poziomu dla akceleratorów GPU. Do utrudnień o których mowa, można zaliczyć także natywność kodu w którym napisane jest źródło. Oznacza to że program wynikowy jest dedykowany na platformę sprzętowo-programową (np. GPU), a więc działa na niej bezpośrednio, bez pomocy innych programów, takich jak emulatory czy maszyny wirtualne, czyli nie można liczyć na wsparcie z takiej platformy. Dodatkowo kod może zrobić się nieczytelny, ponieważ nie można stosować powszechnie już popularnej i efektywnej obiektowości.

Na podstawie uzyskanych wyników czasów, przy różnych rozmiarach przestrzeni automatów komórkowych, można stwierdzić, że zaimplementowane modele na potrzeby niniejszej pracy, są skalowalne przy zastosowaniu ich do obliczeń GPGPU. Można również stwierdzić, że możliwe jest przyspieszenie obliczeń, przy zwiększonej ilości wieloprocesorów strumieniowych. Wyliczone przyspieszenia i skalowalność, na procesorze graficznym Nvidia Tesla M2090, które zostały przeprowadzone w oparciu o zaimplementowane i przedstawione w niniejszej pracy modele mikrostruktur materiałowych, odbiegały od ich idealnych wartości. Mimo to dowiedziono, że możliwe jest uzyskanie przyspieszeń, przeprowadzania symulacji zjawisk mikrostrukturalnych, bazujących na automatach komórkowych. W ten sposób dowiedziono także, iż odpowiednia implementacja modeli materiałowych opartych o automaty komórkowe, na konwencjonalne procesory wielordzeniowe i oraz procesory kart graficznych umożliwia zwiększenie efektywności działania zastosowanych algorytmów.

Na podstawie uzyskanych wyników wydajności obliczeń udowodniono, że warto zainteresować się programowaniem równoległym na akceleratorach GPU, ponieważ przy dobrze dopracowanych modelach materiałowych i pozbyciu się problemu z sekwencyjnym przesyłem i synchronizacją danych, istnieje możliwość wiarygodnej symulacji zjawisk fizycznych zachodzących w mikrostrukturach materiałów. Dzięki wynikom o dużej dokładności złożonych modeli mikrostruktur materiałowych, można odstąpić od kosztownych badań eksperymentalnych i zastąpić je tańszymi symulacjami komputerowymi, pomimo tego że stworzenie optymalnego kodu, okupione jest szeregiem testów. Na korzyść akceleratorów GPU w zestawieniu z procesorami CPU, przemawia także niższy koszt tych pierwszych. Dla przykładu można podać, że koszt[[14]](#footnote-14) karty graficznej Nvidia GTX 650 to ok. 400 zł, a udostępnia ona moc obliczeniową o wartości ok. 813 GFLOPS[[15]](#footnote-15) pojedynczej precyzji, natomiast przykładowy procesor Intel Core i5-2500K, charakteryzuje się wydajnością tylko 110 GFLOPS’ów pojedynczej precyzji, a jego cena wacha się w granicach aż 850 zł. A warto mieć na uwadze, że obecna technologia nie pozwala na zwiększenie częstotliwości ponad około 3 GHz, bez znacznego poboru wzrostu prądu, optymalizacje architektury także mają swoje granice. Obecnie jedyną możliwością znacznego zwiększenia mocy obliczeniowej pojedynczego komputera jest zwiększenie liczby procesorów bądź rdzeni w procesorze.

Na pewno warto skupić się na rozwiązaniu problemu związanego z synchronizacją danych. Jest to główny problem dotyczący technologii GPGPU, ponieważ mimo że obliczenia na kartach graficznych, mogą być bardzo wydajne, to opatrzone są one wysoki kosztem synchronizacji, która stanowi ograniczenie dla uzyskania jeszcze lepszych wyników. Można powiedzieć, że możliwości jakie daje GPGPU, nie są na tyle rewelacyjne, aby można było poprzestać nad ich rozwojem.

Niniejsza praca dowodzi, że warto skupić się na rozwijającej się technologii GPGPU. Do zalet tej technologii należy również to, że będzie ona na pewno rozwijana ze względu na to, że moc obliczeniowa kart graficznych nie jest wykorzystywana tylko w badaniach naukowych, ale i również przemyśle gier komputerowych, który obecnie przynosi na świecie olbrzymie zyski. Ideologia automatów komórkowych jest podobna do interpretacji wartości pojedynczego piksela dla obrazów, ponieważ każda komórka determinowana jest przez te same reguły.

# BIBLIOGRAFIA

[1] Munshi A., Gaster B., Mattson T., Fung J., Ginsburg D., *OpenCL Proramming Guide,* Addison-Wesley 2011

[2] *Introduction to OpenCL Programming – Training Guide ,* ADM, May 2010

[3] *AMD Accelerated Parallel Processing OpenCL, Programming Guide*, AMD, December 2011

[4] *Programming Guide, ATI Stream Computing OpenCL™,* June 2010

[5] Banaś K., Wykłady z przedmiotu *Programowanie Równoległe Wieloskalowe na kierunku Informatyka Stosowana*, Wydział Inżynierii Metali i Informatyki Przemysłowej, AGH 2011/2012 (http://www.metal.agh.edu.pl/~banas/PR/PR.html, data korzystania: listopad 2012)

[6] Przybyłowicz K., *Metaloznawstwo,* Wydawnictwo Naukowo-Techniczne, Warszawa 1999

[7] *Laboratorium nauki o materiałach,* Politechnika Łódzka, Wydział Mechaniczny, Instytut Inżynierii Materiałowej, Łódź 2010

[8] http://www.nvidia.com/docs/IO/105880/DS\_Tesla-M2090\_LR.pdf (data korzystania: 03.11.2012)

[9] http://www.nvidia.com/docs/IO/43395/Tesla-M2090-Board-Specification.pdf (data korzystania: 03.11.2012)

[10] Gawąd J., *Modelowanie ielkoskalowe metodą automatów komórkowych własności materiałów odkształcanych plastycznie,* Rozprawa Doktorska, Promotor: prof. Dr hab. Inż. Maciej Pietrzyk, Kraków 2007

[11] Madej Ł., *A perceptive comparison of the cellular automata and Monte Carlo techniques*

*in application to static recrystallization modeling in polycrystalline materials*, Computational Materials Science 67 (2013) 156–173, AGH University of Science and Technology, 2012

[12] Madej Ł., *Wykłady z przedmiotu Modelowanie Wieloskalowe na kierunku Informatyka Stosowana*, Wydział Inżynierii Metali i Informatyki Przemysłowej, AGH 2011/2012 (http://home.agh.edu.pl/~lmadej/index.php?id=multiscale, data korzystania: listopad 2012)

[13] Nowak K., *Zastosowanie automatów komórkowych w opisie rozwoju uszkodzeń w warunkach pełzania*, Rozprawa Doktorska, Promotor: prof. Dr hab. Inż. Marcin Chranowski, Kraków 2009

[14] Shibutani Y., *Mesoscopic dynamics on dislocation patterning in fatigued material by cellular automata*, Materials Science Research International, Vol.5 No.4, 1999

[15] http://www.cs.cmu.edu/afs/cs/academic/class/15869-f11/www/readings/lindholm08\_tesla.pdf (data korzystania: 10.11.2012)

# Zawartość CD

* Elektroniczna wersja projektu dyplomowego
* Kody źródła do trzech zaimplementowanych modeli mikrostruktur materiałowych
* Aplikacja napisana przez Artura Rodzaja, której zadaniem jest uruchamianie i rozdzielanie przestrzeni pomiędzy procesory CPU i GPU
* Narzędzie do debugowania kody źródeł na procesorach sekwencyjnych

# Spis ilustracji

|  |  |  |
| --- | --- | --- |
| *Rys. 2.1* | ***Praca elementów roboczych w obrębie tej samej grupy i poza nią.*** | *10* |
| *Rys. 2.2* | *Programowanie proceduralne vs. programowanie równoległe* | *10* |
| *Rys. 2.3* | *Etapy tworzenia programy napisanego przy pomocy OpenCL* | *11* |
| *Rys. 2.4* | *„Zestaw narzędziowy” kompilatora OpenCL* | *12* |
| *Rys. 2.5* | *Środowisko uruchomieniowe* | *14* |
| *Rys. 2.6* | *Struktura projektu PL-Grid* | *16* |
| *Rys. 2.7* | *Schemat blokowy używanego w TESLA™ M2090podwójnego gniazda modułu obliczeniowego* | *18* |
| *Rys. 2.8* | *Schemat blokowy używanego w TESLA™ M2090podwójnego gniazda modułu obliczeniowego* | *18* |
| *Rys. 3.1* | *Rozrośnięta przestrzeń, przy zastosowaniu sąsiedztwa: a.) Moore’a,*  *b.) von Neuman’a, c.) Heksagonalne (losowe), d.) pentagonalne.* | *24* |
| *Rys. 3.2* | *Rozrost ziaren przy sąsiedztwie heksagonalnym losowym* | *25* |
| *Rys. 3.3* | *Sąsiedztwo von Neumana* | *27* |
| *Rys. 3.4* | *Rekrystalizacja dynamiczna, która została przeprowadzona na przestrzeni a.), b.) pojawienie się zarodów rekrystalizacji, c) rozrost zarodków, d.) zrekrystalizowana*  *struktura.* | *30* |
| *Rys. 3.5* | *Przestrzeń dla pękania zmęczeniowego z warunkami brzegowymi periodycznymi i zamkniętymi pochłaniającymi* | *34* |
| *Rys. 3.6* | *Pękanie zmęczeniowe na granicy dwóch ziaren.* | *36* |
| *Rys. 4.1* | *Zużycie akceleratorów czterech graficznych GPU, odczytywane co 1 s, dla modelu pękania zmęczeniowego uruchomionego na przestrzeni 200x200, przy 200 iteracjach.* | *43* |
| *Rys. 4.2* | *Zużycie akceleratorów czterech graficznych GPU, odczytywane co 1 s, dla modelu pękania zmęczeniowego uruchomionego na przestrzeni 1000x1000, przy 200 iteracjach.* | *44* |
| *Rys. 4.3* | *Zużycie akceleratorów czterech graficznych GPU, odczytywane co 1 s, dla modelu pękania zmęczeniowego uruchomionego na przestrzeni 200x200, przy 1000 iteracjach* | *45* |

# Spis tabel

|  |  |  |
| --- | --- | --- |
| *Tab. 4.1* | Zestawienie czasu obliczeń trzech zaimplementowanych modeli mikrostruktur materiałowych. | *38* |
| *Tab. 4.2* | *Zestawienie czasu obliczeń dla różnych przestrzeni, przy różnej ilości iteracji na jednym procesorze graficznym GPU* | *39* |
| *Tab. 4.3* | *Zestawienie czasu obliczeń dla różnych przestrzeni, przy różnej ilości iteracji na dwóch procesorach graficznych GPU* | *40* |
| *Tab. 4.4* | *Zestawienie czasu obliczeń dla różnych przestrzeni, przy różnej ilości iteracji na czterech procesorach graficznych GPU.* | *41* |
| *Tab. 4.5* | Zestawienie czasu obliczeń dla różnych przestrzeni, przy różnej ilości iteracji na jednym procesorze CPU. | *46* |
| *Tab. 4.6* | *Czasy obliczeń i przyspieszenie dla różnej ilości wieloprocesorów strumieniowych, model pękania zmęczeniowego.* | *48* |
| *Tab. 4.7* | *Czasy obliczeń i przyspieszenie dla różnej ilości wieloprocesorów strumieniowych, model naiwnego rozrostu ziaren.* | *48* |
| *Tab. 4.8* | Czasy obliczeń i skalowalność dla różnej ilości wieloprocesorów strumieniowych, model pękania zmęczeniowego. | *49* |
| *Tab. 4.9* | Czasy obliczeń i skalowalność dla różnej ilości wieloprocesorów strumieniowych, model rekrystalizacji dynamicznej. | *52* |
| *Tab. 4.10* | Czasy obliczeń i skalowalność dla różnej ilości wieloprocesorów strumieniowych, model naiwnego rozrostu ziaren | *53* |
| *Tab. 4.11* | Czasy obliczeń i skalowalność dla różnej ilości wieloprocesorów strumieniowych, model naiwnego rozrostu ziaren | *54* |

# Spis wykresów

|  |  |  |
| --- | --- | --- |
| *Wyk. 4.1* | Zestawienie czasu obliczeń w sekundach dla różnych przestrzeni, przy różnej ilości iteracji na jednym procesorze graficznym GPU | *39* |
| *Wyk. 4.2* | *Zestawienie czasu obliczeń w sekundach dla różnych przestrzeni, przy różnej ilości iteracji na dwóch procesorach graficznych GPU.* | *40* |
| *Wyk. 4.3* | *Zestawienie czasu obliczeń w sekundach dla różnych przestrzeni, przy różnej ilości iteracji na czterech procesorach graficznych GPU.* | *41* |
| *Wyk. 4.4* | *Zestawienie czasu obliczeń dla różnych przestrzeni, przy różnej ilości iteracji na różnej ilości procesorów graficznych GPU.* | *42* |
| *Wyk. 4.5* | Przyspieszenie w zależności od ilości wieloprocesorów strumieniowych, dla modelu pękania zmęczeniowego. | *50* |
| *Wyk. 4.6* | *Przyspieszenie w zależności od ilości wieloprocesorów strumieniowych, dla modelu rekrystalizacji dynamicznej.* | *50* |
| *Wyk. 4.7* | *Przyspieszenie w zależności od ilości wieloprocesorów strumieniowych, dla modelu naiwnego rozrostu ziaren.* | *51* |
| *Wyk. 4.8* | *Skalowalność w zależności od ilości wieloprocesorów strumieniowych, dla modelu naiwnego rozrostu ziaren* | *52* |
| *Wyk. 4.9* | *Skalowalność dla modelu naiwnego rozrostu ziaren w zależności od ilości wieloprocesorów strumieniowych* | *53* |
| *Wyk. 4.10* | *Skalowalność dla modelu naiwnego rozrostu ziaren w zależności od ilości wieloprocesorów strumieniowych.* | *54* |

# Listing programów

|  |  |  |
| --- | --- | --- |
| *Kod. 3.1* | *naive\_growth.cl* | *23* |
| *Kod. 3.2* | ***dynamic\_recrystallization.cl*** | *28* |
| *Kod. 3.3* | *fatigue\_cracking.cl* | *10* |
| *Kod. 4.1* | *Model.cpp* (program autorstwa Artura Rodzaja, artur.rodzaj@zoho.com) | *47* |

# Dodatek: Implementacja modułu do debugowania źródła kodu, języka C99 dla OpenCL

Etymologia słowa debugowanie (z ang. *bug to robak, insekt)* ma swoje miejsce na Uniwersytecie Harvarda, gdzie w komputerze Mark II utkwiła ćma, która uniemożliwiła jego prawidłowe działanie. Admirał Grace Hopper operację usuwania martwego owada nazwała debugowaniem, czyli odrobaczaniem. Proces debugowania w informatyce, tyczy się narzędzi do wyszukiwania błędów w kodzie. Takie narzędzie daje możliwość wykonywania programów trybie pracy krokowej lub z zastosowaniem tak zwanych pułapek (z ang. breakpoints). Ponadto użytkownik ma możliwość podglądu i edycji zawartości rejestrów i pamięci.

Choć co prawda ATI Stream Computing dostarcza eksperymentalną [4] jak na razie możliwość używania debugera GDB (na licencji GNU), do debugowania kodu uruchamianego na x86 CPU, pod systemem Linuks i debugera cygwin/minGW dla systemu Windows. Mimo to źródło programu napisanego w OpenCL, jest trudne do debugowania, dlatego na potrzeby niniejszej pracy zostało napisane narzędzie w środowisku programistycznym NetBeans, pod systemem Linuks, które daje możliwość debogowania kodu przy pomocy środowiska NerBeans. To już w intencji programisty pozostaje to, aby kod napisany w tym narzędziu pozostawał zgodny, ze standardem języka C99. Pozostałe niuanse, takie jaka na przykład zmienna typu uint2, występująca w środowisku OpenCL, lecz nie standardowa dla języka C/C++, została zaimplementowana za pomocą struktury. Zaimplementowane narzędzie odwzorowuje również działanie architektury w ten sposób, że daje możliwość korzystania ze wszystkich funkcji występujących w niej, których działanie daje ten sam wynik.

1. http://www.pcworld.pl/news/88941/ENIAC.to.juz.60.lat.html [↑](#footnote-ref-1)
2. Architektura komputerowa według klasyfikacji Flynna, w której to przetwarzane jest wiele strumienie danych, przez pojedynczy strumień rozkazów. [↑](#footnote-ref-2)
3. Program gospodarza w niniejszej pracy inżynierskiej jest rozumiany jako aplikacja napisana w C++, korzystająca z biblioteki OpenCL. [↑](#footnote-ref-3)
4. dane o danych, np. klasyczne katalogi biblioteczne [↑](#footnote-ref-4)
5. Low Level Virtual Machine (LLVM) – kompilator napisany C++, pierwotnie zaprojektowany dla języka C/C++, później rozwinięty poprzez nakładki do obsługi innych języków [↑](#footnote-ref-5)
6. AMD/ATI Compute Abstraction Layer (CAL)– biblioteki sterowników urządzenia będące interfejsem dla kart graficznych ATI, które umożliwiają sterowaniem bezpośrednim jądrem, za pomocą języka pseudo asemblerowego - ATI Intermediate Language (IL) [↑](#footnote-ref-6)
7. Grupa połączonych jednostek komputerowych, które współpracują ze sobą w celu udostępnienia zintegrowanego środowiska pracy. [↑](#footnote-ref-7)
8. http://projekt.plgrid.pl [↑](#footnote-ref-8)
9. https://portal.plgrid.pl [↑](#footnote-ref-9)
10. http://www.windowsfordevices.com/c/a/News/Nvidia-Tesla-M2090/ [↑](#footnote-ref-10)
11. rodzaj defektu punktowego sieci krystalicznej, nie obsadzony atomem lub jonem węzeł sieci [↑](#footnote-ref-11)
12. http://www.doc.ic.ac.uk/~dt10/research [↑](#footnote-ref-12)
13. http://ark.intel.com/products/47922/Intel-Xeon-Processor-X5650-%2812M-Cache-2\_66-GHz-6\_40-GTs-Intel-QPI%29 [↑](#footnote-ref-13)
14. Stan na grudzień 2012 [↑](#footnote-ref-14)
15. http://www.nordichardware.com/Graphics/nvidia-round-off-kepler-with-geforce-gtx-650-ti.html [↑](#footnote-ref-15)