ARTHUR ARANTES FARIA

PROGRAMAÇÃO PARALELA EM CUDA

LAVRAS – MG

2018

ARTHUR ARANTES FARIA

PROGRAMAÇÃO PARALELA EM CUDA

Relatório sobre a API de programação CUDA desenvolvida pela NVIDIA para computação em unidades de processamento gráfico, apresentado na disciplina de Projeto e analise de algoritmos da Universidade Federal de Lavras como parte das exigências para obtenção de nota na disciplina.

LAVRAS – MG

2018

SUMÁRIO

1 INTRODUÇÃO

Sistemas computacionais normalmente podem ser divididos em partes distintas que podem ser processadas ao mesmo tempo, consequentemente podem ter diversas instruções executando paralelamente. Além disso, técnicas de paralelismo podem ser utilizadas em diversos problemas computacionais.

O hardware tem evoluído rapidamente quando comparado ao desenvolvimento de software. Com o objetivo de melhorar o desempenho dos sistemas de modo geral, surgiram as arquiteturas de múltiplos núcleos (*multicore*), que unem em um mesmo processador múltiplas unidades de processamento para a realizar operações computacionais. Uma forma eficiente de explorar essas arquiteturas *multicore* é a contrução de aplicações com paralelismo utilizando recursos que permitam o desenvolvimento de programas com vários fluxos de processos com execução concorrentes (*threads*).

Contudo, deve ser avaliado se é interessante aplicar o paralelismo em determinadas situações. De fato, o processamento paralelo tem se tornado mais difundido, porém a programação específica sobre essas plataformas e a identificação de problemas que se beneficiam de paralelismo ainda não é amplamente vista.

Um grande avanço na área da programação paralela foi a ferramenta desenvolvida pela NVIDIA, a interface de programação (*Application Programming Interface - API*) denominada CUDA (*Computer Unified Device Architecture*), que diferentemente das outras APIs, como o OpenMP, a da NVIDIA utiliza o processamento da GPU (Graphics Processing Units), que previamente era utilizada apenas em tarefas gráficas em três dimensões, para completar grandes processos aritméticos. CUDA é uma arquitetura de computação paralela e um modelo de programação que aumenta o desempenho computacional ao fazer o uso da unidade de processamento gráfico.

O relatório tem como objetivo o estudo da programação paralela com plataforma CUDA. Para isso foram escolhidos problemas computacionais que possam se beneficiar do processamento paralelo, entre eles, a multiplicação de matrizes e o cálculo da sequência de Fibonacci.

2 MATERIAIS E MÉTODOS

Nesta seção serão abordados conceitos sobre o que é necessário para a compressão deste relatório, tendo o objetivo de apresentar as ferramentas: CUDA, OpenMP, OpenCL. Por fim, serão vistos conceitos que envolvem a programação, como uma panorama geral sobre Processos e Programação paralela.

2.1 Processos

Um processo pode ser definido como uma das entidades de um programa que possui um contador que determina a próxima instrução a ser executada (SILBERSCHATZ; GALVIN; GAGNE, 2000). No geral, pode ser considerado um programa em execução, ou seja, está sendo executado em um dos processadores virtuais do sistema operacional (TANENBAUM; STEEN, 2008). Entretanto, o programa de um computador pode ser composto por uma série de processos.

A capacidade de um sistema operacional de executar simultaneamente dois ou mais processos é chamado de multiprocessamento. Tanenbaum e Steen (2008) expõe que em um computador são executados vários processos de maneira concorrente, o sistema operacional monitorara as atividades, assim como o gerenciamento de memória.

Os processos quando executados, passam por uma série de fases, cada fase é chamado de estado. De acordo com Silberschatz, Galvin e Gagne (2000) um processo executado pode estar nos seguintes estados:

- **Novo:** estado em que o processo é criado;
- **Em execução:** execução das instruções;
- **Em espera:** esperando por algum evento;
- **Pronto:** o processo está pronto para ser executado por um processador;
- **Terminado:** o processo é encerrado.

Com uma arquitetura que proporciona o multiprocessamento, o desenvolvedor tem o potencial de implementar multiprogramação. A multiprogramação tem como objetivo garantir a execução constante de processos concorrentes de forma a maximizar a utilização da CPU (SILBERSCHATZ; GALVIN; GAGNE, 2000).

Silberschatz, Galvin e Gagne (2000) também mostram que existem duas modalidades de processos concorrentes: aqueles que não afetam a execução de outros processos, isto é, que podem não compartilhar dados com outros processos, estes são chamados de processos independentes; e aqueles onde a execução afeta os outros processos e que por ventura possa a vir compartilhar dados com os outros processos, são chamados de processos cooperativos.

Um ambiente que proporciona processos cooperativos possui muitas vantagens. Essas características vão ser identificadas durante o desenvolvimento do trabalho, pois o problema estudado envolve o compartilhamento de informações, processamento mais rápido com a divisão de tarefas em sub tarefas e modularidade. A modularidade refere-se à divisão das funções do programa em processos separados (SILBERSCHATZ; GALVIN; GAGNE, 2000).

Além disso, Silberschatz, Galvin e Gagne (2000) mostra que baseado no fato de que um processo é uma unidade de trabalho em um sistema, pode-se implicitamente descrever que cada processo em um determinado momento executa uma de suas atividades para realizar uma ação específica. Essas ações são chamadas de threads, as quais são usadas também para o controle de execução dos processos.

Silberschatz, Galvin e Gagne (2000) definie um processo leve (LWP – *light-weight process*) ou simplesmente thread como uma unidade de utilização da CPU composta por um ID que compartilha com outras threads do mesmo processador os recursos deste, como seção de código. Quando se gerencia múltiplas threads, a aplicação deverá gerenciar de forma adequada o acesso de dados para não ocorrer *deadlocks* - situação em que ocorre um impasse, e dois ou mais processos ficam impedidos de continuar suas execuções. Essas situações acontecem quando uma thread permanece bloqueada no aguardo de ums inal de outra (que nunca é enviado), ou seja, falha na comunicação e sincronização dos processos leves.

2.2 Programação paralela

Tipicamente, *softwares* tem sido desenvolvidos para serem executados sequencialmente. Para resolver um problema, um algoritmo é construído e implementando um fluxo sequencial de instruções. Tais instruções são então executadas por um CPU. Somente uma instrução pode ser executada por vez; após sua execução, a próxima então é executada (BARNEY et al., 2010).

Porém, a computação paralela faz uso de múltiplos elementos de processamento simultâneo para resolver um problema em questão. Com isso, é possível quebrar um problema em diversas partes independentes, assim, de tal forma cada elemento do processamento poderá exe-

cutar sua parte do algoritmo simultaneamente com os outros. Os elementos de processamento podem ser diversos e incluir recursos como um único computador com múltiplos processadores, diversos computadores em rede, *hardware* especializado ou qualquer combinação dos anteriores (BARNEY et al., 2010).

O aumento da frequência de processamento, foi o principal motivo para melhorar o desempenho dos computadores no inicio da década de 1990. Mantendo todo o restante constante, aumentar a frequência de processamento de um computador tende a reduzir o tempo médio para executar uma instrução, reduzindo então o tempo de execução para todos os programas que demandam uma alta taxa de processamento (HENNESSY; PATTERSON, 2011).

Todavia, um dos objetivos de usar a paralelização é aumentar o desempenho da aplicação com a execução nos vários núcleos disponíveis na máquina. Existem na literatura duas abordagens disponíveis de paralelismo: a autoparalelização e programação paralela. A primeira delas, a autoparalelização, ocorre quando uma aplicação é desenvolvida sequencialmente não modelada para ser processada paralelamente, porém o compilador tenta automaticamente paralelizar essa aplicação para que possa se aproveitar da estrutura do hardware (e.x.: *multicore*, *gpu*). Já a segunda delas, a programação paralela desenvolve o paralelismo desde o algoritmo, logo, está abordagem oferece maior ganho de desempenho, porém exige um esforço pleno do desenvolvedor em sua elaboração.

2.3 CUDA

CUDA é uma API de programação paralela criada pela NVIDIA com o objetivo de melhorar o desempenho computacional através do uso do processamento gráfico da GPU. Com essa API as placas de vídeo que antes eram restritas apenas para a computação gráfica podem ser usadas para propósitos gerais. CUDA referencia a CPU como *host* e a GPU como *device*.

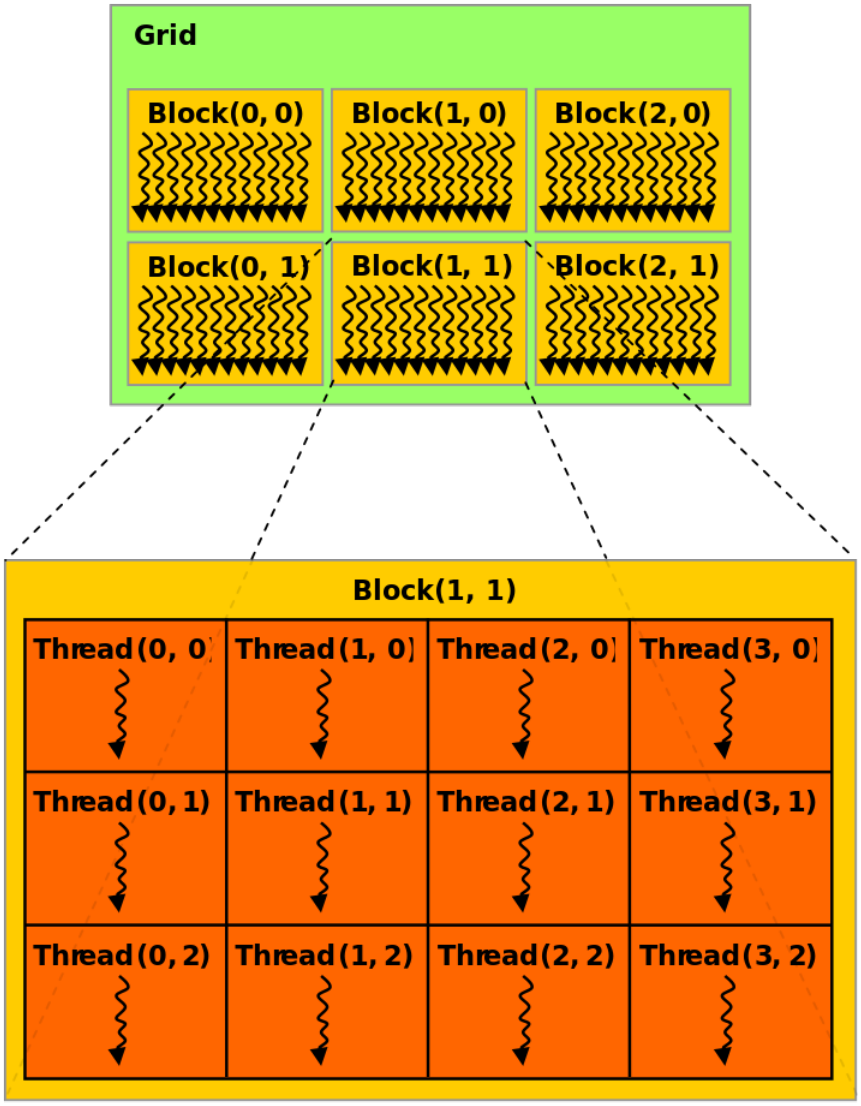
Uma GPU que tem a possibilidade de programação com CUDA pode possuir centenas de núcleos (*cores*) onde podem ser executados até milhares de *threads* (SANDERS; KANDROT, 2010). Essa arquitetura é composta por Streaming Multiprocessors (SMs), de acordo com Kirk e Wen-Mei (2016) a quantidade contida (de SMs) na placa de vídeo varia de uma geração para outra de GPUs da NVIDIA.

Em CUDA, um bloco é uma unidade de organização e mapeamento de *threads* sobre o hardware, cada um destes blocos são alocados a um multiprocessador na GPU. Uma grade é

uma estrutura onde se definem um conjunto de blocos e *threads* que serão criados e gerenciados pela GPU para uma determinada função pré programada.

Os blocos (*blocks*) de *threads* são organizados em grades (*grids*). Os blocos podem ser unidimensionais, vetores, bidimensionais, matrizes. A Figura 2.2 expõe uma representação de um *grid* de tamanho 3x2 contendo blocos de tamanho 4x3. Tantos os blocos quanto as *threads* possuem sua identificação própria.

Figura 2.1 – Exemplo de grid, blocks e threads. Disponivel em: (CUDA, 2013)

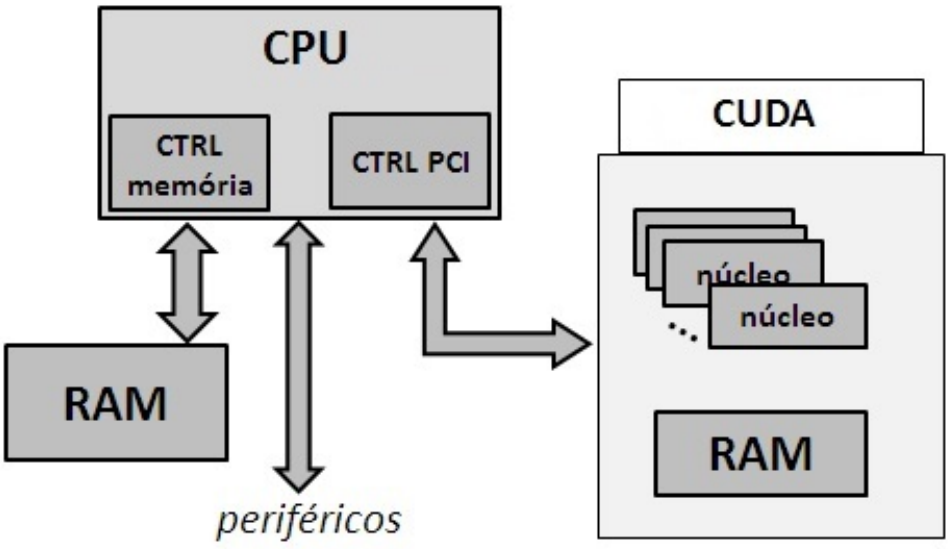


De acordo com Franco et al. (2009) em CUDA, a programação na GPU é realizada através de uma função chamada no *kernel*, essa função é executada na máquina por várias *threads* que rodam em diferentes processadores da placa gráfica. Em um bloco, contem-se *threads* que trocam de maneira eficiente os dados através do compartilhamento de memória e

a sincronização desse bloco durante a execução. Cada bloco de *thread* é executado em um multiprocessador e cada *thread* possui uma identificação (FRANCO et al., 2009).

A chamada do *kernel* é realizada pela CPU. Todos os dados e configurações são inicializados pela CPU e copiados para a memória da GPU, após a GPU terminar as atividades, os resultados devem ser retornados para a memória principal. A Figura 2.2 ilustra o relacionamento dos principais componentes do computador. A transferência de dados entre CPU e GPU, muita das vezes, é realizada através de uma tecnologia chamada memória unificada, o que deixa esse processo transparente.

Figura 2.2 – Relacionamento entre CPU e GPU CUDA



No *kernel*, é percebido que a programação paralela em CUDA se torna simplificada, porque não há necessidade de escrever códigos com *threads* tradicionais. Os *kernels* são executados em paralelo. As *threads* contidas dentro dos blocos se comunicam através do compartilhamento extremamente rápido da memória, essa otimização no acesso da memória é fundamental para um bom desempenho da GPU (SUCHARD et al., 2010).

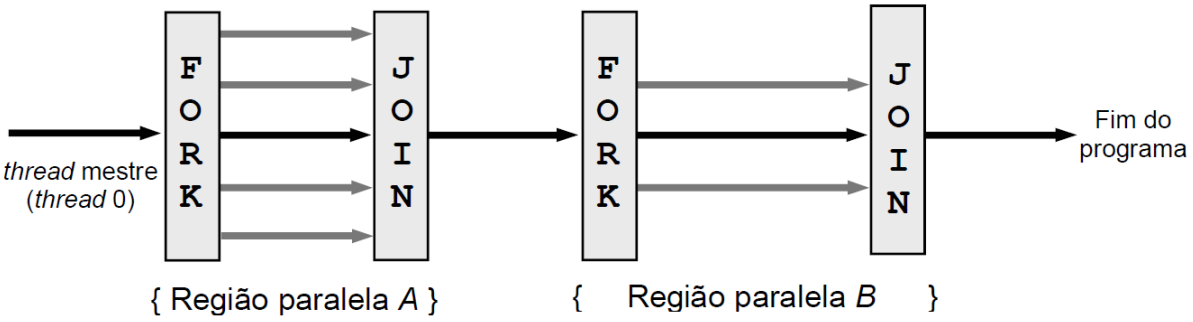
Em resumo, a sequência de execução básica de um algoritmo em CUDA segue os seguintes passos: os dados são processados e inicializados pelo *host*, em seguida os dados processados são copiados para a memória do *device* que após o processamento (execução do *kernel* para as tarefas paralelas), são retornados para a memória principal.

2.4 OpenMP

O OpenMP é um padrão criado e mantido por uma organização independente (openmp.org) que especifica um conjunto de diretivas de compilação, funções e variáveis de ambiente para a programação paralela, disponível em arquiteturas com memória compartilhada nas linguagens Fortran 77, Fortran 90, C e C++ (QUINLAN et al., 2003). Ao compilar um programa em Fortran ou C++ contendo diretivas OpenMP com um compilador que implementa as diretivas OpenMP, o programador irá obter um programa em que será executado em paralelo, com as diretivas implementadas.

Um programa que utiliza do OpenMP inicia com uma única *thread*, a *thread* mestre, será executada até encontrar uma região paralela. Neste ponto é criado outras *threads*. Então, todas as *threads*, inclusive a mestre, executam o código dentro desta região. Quando todas as *threads* completam a execução do código da região paralela elas se sincronizam, logo após a exceção da *thread* mestre. A execução então, continua sequencialmente com a *thread* mestre até que ela encontre uma nova região paralela ou até que o algoritmo termine. Este é o modelo de execução conhecido como *fork-join* como mostra a Figura 2.3

Figura 2.3 – Modelo de execução do OpenMP. Exemplo de um programa com regiões paralelas, A e B.



A criação das *threads* em um programa OpenMP é responsabilidade do compilador. O programador precisa apenas definir, através de diretivas. Uma diretiva consiste de uma linha de código com significado especial para o compilador, no caso o OpenMP, suas diretivas indicam as regiões aonde ele é utilizado. O OpenMP oferece duas diretivas para definição de regiões paralelas, uma delas é utilizada para a definição de regiões contendo laços *for*. Portanto, para paralelizar um laço *for* define-se uma região paralela e, dentro dela, o laço. Neste relatório, não será abordado a fundo sobre todas as diretivas disponíveis por não ser o foco principal do mesmo.

2.5 OpenCL

O OpenCL é um padrão aberto, que permite o uso de GPUs para desenvolvimento de aplicações paralelas. A API também permite que os desenvolvedores escrevam códigos de programação heterogêneos, fazendo com que estes programas consigam aproveitar tanto os recursos de processamento das CPUs quanto das GPUs (TSUCHIYAMA et al., 2010).

Apesar de se tratar de um sistema aberto, há um grupo responsável pela padronização de alguns parâmetros. O grupo anunciou recentemente uma versão atualizada do OpenCL. O OpenCL 2.2 é a mais recente evolução do padrão OpenCL, projetado para simplificar a programação multiplataforma, permitindo uma variedade de algoritmos e padrões de programação para serem facilmente acelerados. Ainda visa fornecer uma série de vantagens interessantes para os desenvolvedores (GROUP., 2018).

Os dispositivos em OpenCL podem ou não compartilhar memória com a CPU e, normalmente, têm um conjunto de instruções de máquina diferente (STONE; GOHARA; SHI, 2010). A APIs fornecida pelo OpenCL, para cada versão, inclui funções para enumerar os dispositivos disponíveis (CPU, GPU e outros aceleradores), gerenciar as alocações de memória, realizar transferências de dados entre CPU e GPU, chamar *kernels* para serem executados nos núcleos da GPU e também verificar erros.

Após seu surgimento, alguns trabalhos têm utilizado o OpenCL na tentativa de aumentar o desempenho computacional de problemas paralelizável. Entre eles, pode-se citar o trabalho de Komatsu et al. (2010), que apresentaram uma avaliação de desempenho e portabilidade de programas em OpenCL. Barak et al. (2010) apresentaram algumas aplicações em OpenCL executadas em *clusters* com muitas GPUs. Neste relatório será explorado sobre a comparação entre o tempo de execução de algortimos com OpenCL e OpenMP e principalmente comparado com CUDA.

3 TRABALHOS RELACIONADOS

O uso das GPUs como propósito geral tem se tornado constante. Muitos pesquisadores têm feito a comparação de desempenho entre o uso das CPUs e GPUs, como Pospichal e Jaros (2009), que usou as funções de Rosenbrock, Griewank e Michalewicz.

O trabalho realizado por Lee et al. (2010) trata exatamente dessa comparação. Os autores apresentam uma análise da programação paralela aplicando técnicas de otimização sobre CPU e GPU. Este estudo baseia-se na comparação de vários problemas diferentes. As aplicações escolhidas possuem um nível de paralelismo elevado, que por natureza podem ser modeladas sobre as arquiteturas multicores moderna. Na pesquisa, concluíram que *multithreading*, *cache blocking* e reorganização de acesso à memória SIMD (*Single Instruction, Multiple Data*) são técnicas que possuem um melhor desempenho na CPU; já para GPU especifica como minimização da sincronização global e *buffers* localmente compartilhados como as duas técnicas chaves de melhor desempenho.

Como foi citado nos capítulos anteriores, CUDA utiliza blocos para mapeamento de *threads* na GPU. A comunicação das *threads* que ocorrem dentro do bloco ocorre na memória global, e requer uma sincronização (XIAO; FENG, 2010). Essa operação é realizada apenas na GPU e causa uma sobrecarga que compromete o desempenho da aplicação. Baseado nisso, os autores do trabalho citado propõem métodos mais eficientes para essa sincronização, apresentando um modelo de desempenho para a execução dos *kernels*. Este exemplo mostra como o conhecimento da arquitetura e a organização do código pode ter um efeito muito importante na velocidade de execução.

Mesmo não sendo utilizado nestes relatório, devido a um inviabilidade técnica em número de computadores disponíveis, é valido citar que uma outra maneira de explorar a arquitetura CUDA é usando placas GPU em um *cluster* (conjunto de computadores interconectados que funcionam como se fosse um único grande sistema); um exemplo nesse sentido é apresentado por Asunción et al. (2012) onde ele fez uso quatro computadores conectados em rede. A comunicação entre os computadores nesse caso utilizou MPI (*Message Passing Interface*).

Finalmente, outras arquiteturas de GPU também podem ser exploradas para paralelismo. Exemplificando, o trabalho de Du et al. (2012) cria uma ferramenta para gerar código OpenCL ou Cuda, de maneira transparente para o programador que deseja empregar placas gráficas que usem um desses padrões.

4 EXPERIMENTOS

Neste capitulo será exposto os resultados dos experimentos utilizados para resolução do exemplo proposto para os testes, a fim de demonstrar o uso da ferramenta em questão. Nele também será apresentado os resultados com a plataforma CUDA, e das outras ferramentas supramencionadas, através do desenvolvimento das tarefas específicas propostas. Sendo elas: multiplicação de matrizes e o cálculo da sequência de Fibonacci. Esses algoritmos possibilitaram uma visão de o quanto bem cada ferramenta trabalha com o paralelismo.

Foi utilizado como métrica de performance o tempo de relógio de execução de cada algoritmo, os testes foram executados em um computador pessoal com as seguintes configurações: Processador Intel® Core™ i7-6700HQ *cache* de 6M, *clock* até 3,50 GHz; 16gb ram 2133 *MHz* DDR4; placa de video GeForce GTX 970M com 1280 CUDA *cores* e 3gb de memoria ram dedicada.

A Tabela 4.1 mostra os resultados para a execução do algoritmo que realiza o calculo da sequencia de Fibonnaci para um intervalo de valores. Vale ressaltar que o algoritmo implementado foi uma versão recursiva, pois nesse caso, tem um tempo de execução maior que sua versão interativa.

Tabela 4.1 – Tempo de execução em segundos para a sequencia de Fibonnaci

n	**Sequencial**	**OpenMP**	**CUDA**
50	276.221968	273.421303	**0.000000**

A Tabela 4.2 ilustra os resultados para a execução do algoritmo de multiplicação de matrizes. O parâmetro utilizado foi uma matriz quadrada de tamanho $n = 100; 1000; 10000$.

Tabela 4.2 – Tempo de execução em segundos para multiplicação de matrizes

n	**Sequencial**	**OpenMP**	**CUDA**
50	0.002010	0.000000	**0.000034**
500	0.929516	0.902617	**0.000046**
1000	7.933796	7.931795	**0.000057**

5 CONCLUSÃO

Após todo o estudo das ferramentas descritas no Capítulo 2 e com os teste realizados expostos no Capítulo 4 esta seção apresenta as considerações finais do trabalho fazendo uma relação dos resultados obtidos com o exposto de métodos e materiais.

Com os testes executados, pode-se perceber uma vantagem clara em ganho de tempo de execução de um algoritmo quando é executado na GPU utilizando CUDA, quando comparado com os outros meios de execução.

Acreditasse que a vantagem clara que a GPU tem diante a CPU é a quantidade de núcleos, logo, quanto mais núcleos a placa tiver, maior poder de processamento e melhor seu o desempenho será. Por outro lado, a vantagem da CPU é o seu relacionamento direto com os outros componentes do computador por ser um componente central indispensável.

REFERÊNCIAS

ASUNCIÓN, M. D. L. et al. An mpi-cuda implementation of an improved roe method for two-layer shallow water systems. **Journal of Parallel and Distributed Computing**, Elsevier, v. 72, n. 9, p. 1065–1072, 2012.

BARAK, A. et al. A package for opencl based heterogeneous computing on clusters with many gpu devices. In: IEEE. **Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS)**. IEEE International Conference, 2010. p. 1–7.

BARNEY, B. et al. Introduction to parallel computing. **Lawrence Livermore National Laboratory**, v. 6, n. 13, p. 10, 2010.

CUDA, C. **Programming guide NVIDIA Corporation**. .: July, 2013.

DU, P. et al. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. **Parallel Computing**, Elsevier, v. 38, n. 8, p. 391–407, 2012.

FRANCO, J. et al. A parallel implementation of the 2d wavelet transform using cuda. In: IEEE. **Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on**. Weimar, Germany, 2009. p. 111–118.

GROUP., K. **Thread Hierarchy in CUDA Programming**. 2018. Disponível em: <https://www.khronos.org/opencl/>.

HENNESSY, J. L.; PATTERSON, D. A. **Computer architecture: a quantitative approach**. Waltham, USA: Elsevier, 2011.

KIRK, D. B.; WEN-MEI, W. H. **Programming massively parallel processors: a hands-on approach**. Cambridge: Morgan kaufmann, 2016.

KOMATSU, K. et al. H.: Evaluating performance and portability of opencl programs. In: **Proc. Automatic Performance tuning**. [S.l.: s.n.], 2010.

LEE, V. W. et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. **ACM SIGARCH computer architecture news**, ACM, v. 38, n. 3, p. 451–460, 2010.

POSPICHAL, P.; JAROS, J. Gpu-based acceleration of the genetic algorithm. **GECCO competition**, Citeseer, 2009.

QUINLAN, D. et al. A c++ infrastructure for automatic introduction and translation of openmp directives. In: SPRINGER. **International Workshop on OpenMP Applications and Tools**. Berlin, Heidelberg., 2003. p. 13–25.

SANDERS, J.; KANDROT, E. **CUDA by example: an introduction to general-purpose GPU programming**. USA: Addison-Wesley Professional, 2010.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Fundamentos de Sistemas Operacionais .** São Paulo: Grupo Gen-LTC, 2000.

STONE, J. E.; GOHARA, D.; SHI, G. Opencl: A parallel programming standard for heterogeneous computing systems. **Computing in science & engineering**, IEEE, v. 12, n. 3, p. 66–73, 2010.

SUCHARD, M. A. et al. Understanding gpu programming for statistical computation: Studies in massively parallel massive mixtures. **Journal of computational and graphical statistics**, Taylor & Francis, v. 19, n. 2, p. 419–438, 2010.

TANENBAUM, A. S.; STEEN, M. V. **Sistemas distribuidos: principios e paradigmas**. 2. ed. São Paulo: Pearson, 2008.

TSUCHIYAMA, R. et al. The opencl programming book. **Fixstars Corporation**, Japan, v. 63, 2010.

XIAO, S.; FENG, W.-c. Inter-block gpu communication via fast barrier synchronization. In: IEEE. **Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on**. Atlanta, GA, USA, 2010. p. 1–12.