INSTITUTO DE ENGENHARIA NUCLEAR

SÉRGIO RICARDO DOS SANTOS MORAES

COMPUTAÇÃO PARALELA EM *CLUSTER* DE GPU APLICADO A PROBLEMA DA ENGENHARIA NUCLEAR

Rio de Janeiro
2012

SÉRGIO RICARDO DOS SANTOS MORAES

COMPUTAÇÃO PARALELA EM *CLUSTER* DE GPU APLICADO A PROBLEMA DA ENGENHARIA NUCLEAR

Dissertação apresentada ao Programa de Pós-Graduação em Ciência e Tecnologia Nucleares do Instituto de Engenharia Nuclear da Comissão Nacional de Energia Nuclear como parte dos requisitos necessários para a obtenção do Grau de Mestre em Ciências em Engenharia Nuclear – Profissional em Engenharia de Reatores

Orientador: Prof. Dr. Cláudio Márcio do Nascimento A. Pereira
Co-orientador: Prof. Dr. Antônio Carlos de Abreu Mól

Rio de Janeiro
2012

MORA Moraes, Sérgio Ricardo dos Santos
Computação paralela em cluster de gpu aplicado a problema da engenharia nuclear / Sérgio Ricardo dos Santos Moraes. – Rio de Janeiro: CNEN/IEN, 2012.

109f.

Orientadores: Dr. Cláudio Márcio do Nascimento A. Pereira
e Dr. Antônio Carlos de Abreu Mól
Dissertação (mestrado) – Instituto de Engenharia Nuclear, PPGIEN, 2012

1. Computação Paralela. 2. GPU. 3. CUDA. 4. Cluster 5. GPU. 6. Thread. 7. Método Monte Carlo. I. Título

CDD

COMPUTAÇÃO PARALELA EM *CLUSTER* DE GPU APLICADO A PROBLEMA DA ENGENHARIA NUCLEAR

Sérgio Ricardo dos Santos Moraes

DISSERTAÇÃO SUBMETIDA AO PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA E TECNOLOGIA NUCLEARES DO INSTITUTO DE ENGENHARIA NUCLEAR DA COMISSÃO NACIONAL DE ENERGIA NUCLEAR COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA NUCLEAR – PROFISSIONAL EM ENGENHARIA DE REATORES

Aprovada por

Prof. Cláudio Márcio do Nascimento A. Pereira, D.Sc.

Prof. Antonio Carlos de Abreu Mol, D.Sc.

Prof. Celso Marcelo Franklin Lapa, D.Sc

Prof. Rafael Pereira Baptista, D.Sc.

RIO DE JANEIRO, RJ – BRASIL

ABRIL DE 2012

Aos meus pais, Olympio e Ray (in memoriam),

e aos meus amores,

Caio e Cleide.

AGRADECIMENTOS

Primeiramente agradeço a Deus, por mais uma conquista em minha vida.

Ao IEN – Instituto de Engenharia Nuclear/CNEN, pela oportunidade de realizar este trabalho.

Aos professores do IEN, que contribuíram para minha formação acadêmica e amadurecimento pessoal.

Em especial, ao Professor Orientador Cláudio Márcio do Nascimento A. Pereira, por ter acreditado em mim, pela sua competência, atenção e disponibilidade em sempre ajudar.

Ao funcionário Adino Américo Heimlich Almeida, pela disponibilidade em sempre ajudar e partilhar seu conhecimento e experiência.

Aos bolsistas do LIAA-IEN, especialmente Pedro Resende, jovem muito capaz, pessoa fundamental para que este trabalho fosse concluído.

Aos colegas de turma dos anos 2009 e 2010, pelos agradáveis momentos de convivência, apoio, incentivo e troca de ideias e experiências.

Aos amigos Isaque Rodrigues e Leandro Gatto, pela paciência em ouvir, pela amizade e pela companhia.

À Professora Ligia Alves dos Santos Souza e à Revisora Lara Alves, amigas sempre solícitas, dedicadas e pacientes em ajudar na concretização deste trabalho.

“Cada sonho que você deixa pra trás
é um pedaço do seu futuro que deixa de existir”.
Steve Jobs

RESUMO

A computação em cluster tem sido amplamente utilizada como uma alternativa de relativo baixo custo para processamento paralelo em aplicações científicas. Com a utilização do padrão de interface de troca de mensagens (MPI, do inglês Message-Passing Interface), o desenvolvimento tornou-se ainda mais acessível e difundido na comunidade científica. Uma tendência mais recente é a utilização de Unidades de Processamento Gráfico (GPU, do inglês Graphic Processing Unit), que são poderosos coprocessadores capazes de realizar centenas de instruções ao mesmo tempo, podendo chegar a uma capacidade de processamento centenas de vezes a de uma CPU. Entretanto, um microcomputador convencional não abriga, em geral, mais de duas GPUs. Portanto, propõe-se neste trabalho o desenvolvimento e avaliação de uma abordagem paralela híbrida de baixo custo na solução de um problema típico da engenharia nuclear. A ideia é utilizar a tecnologia de paralelismo em *clusters* (MPI) em conjunto com a de programação de GPUs (CUDA, do inglês Compute Unified Device Architecture) no desenvolvimento de um sistema para simulação do transporte de nêutrons, através de uma blindagem por meio do Método Monte Carlo. Utilizando a estrutura física de cluster composto de quatro computadores com processadores *quad-core* e 2 GPUs cada, foram desenvolvidos programas utilizando as tecnologias MPI e CUDA. Experimentos empregando diversas configurações, desde 1 até 8 GPUs, foram executados e comparados entre si, bem como com o programa sequencial (não paralelo). Observou-se uma redução do tempo de processamento da ordem de 2.000 vezes quando se comparada a versão paralela de 8 GPUs com a versão sequencial. Os resultados aqui apresentados são discutidos e analisados com o objetivo de destacar ganhos e possíveis limitações da abordagem proposta.

Palavras-chave: Computação paralela, GPU, CUDA, MPI, Método de Monte Carlo, transporte de nêutrons, blindagem.

ABSTRACT

Cluster computing has been widely used as a low cost alternative for parallel processing in scientific applications. With the use of Message-Passing Interface (MPI) protocol development became even more accessible and widespread in the scientific community. A more recent trend is the use of Graphic Processing Unit (GPU), which is a powerful co-processor able to perform hundreds of instructions in parallel, reaching a capacity of hundreds of times the processing of a CPU. However, a standard PC does not allow, in general, more than two GPUs. Hence, it is proposed in this work development and evaluation of a hybrid low cost parallel approach to the solution to a nuclear engineering typical problem. The idea is to use clusters parallelism technology (MPI) together with GPU programming techniques (CUDA – Compute Unified Device Architeture) to simulate neutron transport through a slab using Monte Carlo method. By using a cluster comprised by four quad-core computers with 2 GPU each, it has been developed programs using MPI and CUDA technologies. Experiments, applying different configurations, from 1 to 8 GPUs has been performed and results were compared with the sequential (non-parallel) version. A speed up of about 2.000 times has been observed when comparing the 8-GPU with the sequential version. Results here presented are discussed and analysed with the objective of outlining gains and possible limitations of the proposed approah.

Keywords: Parallel computing, GPU, CUDA, MPI, Monte Carlo Method, neutron transport

LISTA DE ILUSTRAÇÕES

LISTA DE GRÁFICOS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

- ALU Arithmetic Logic Unit
- APU Accelerated Processing Units
- CUDA Compute Unified Device Architecture
- IBM International Business Machines
- IEEE Institute of Electrical and Electronics Engineers
- GPU Graphics Processor Unit
- GPGPU General-Purpose Computation on GPU
- HPC High-Performance Computing
- LIAA Laboratório de Inteligência Artificial
- MPI Message Passing Interface
- MPP Massively Parallel Processors
- NEC Nippon Electric Company, Limited
- ORNL Oak National Laboratory
- PVM Parallel Virtual Machine
- SGI Silicon Graphics International

SUMÁRIO

1 INTRODUÇÃO

Embora os microprocessadores estejam ficando cada vez mais rápidos a cada geração, as demandas a eles impostas estão crescendo no mínimo com a mesma rapidez (TANENBAUM, 2001). É fato que a velocidade dos microprocessadores não pode aumentar indefinidamente, dadas diversas limitações físicas, como por exemplo a velocidade da luz e dissipação de calor. Entretanto, as demandas continuam crescendo. A redução do tamanho dos transistores chegará ao ponto em que cada transistor terá um número tão pequeno de átomos dentro dele que os efeitos da mecânica quântica, como, por exemplo, o princípio da incerteza de Heisenberg, podem se tornar um grande problema (TANENBAUM, 2001). Este contínuo impulso pela busca de melhores desempenhos pelos fabricantes de *software* e *hardware* demanda mais investimentos em pesquisas, criando um ciclo positivo para os usuários com redução de custos, novas interfaces e geração de resultados mais rápidos.

É verdade que essa busca por melhor desempenho sempre existiu. Pode-se citar que, em 1964, a IBM lançou o /360 com emprego de tecnologias avançadas, para a época, como a de *pipeline*, entre outros fabricantes e, em meados da década de 1970, os fabricantes já pesquisavam maneiras de aumentar o desempenho dos sistemas de computação, predominantemente nos *mainframes*. Com o lançamento da família de processadores Intel x86, projetistas já vislumbravam o uso de processamento concorrente e embrionariamente paralelo para, junto com o aumento da frequência do *clock*, obter um maior desempenho dos processadores.

A grande maioria dos programas é escrita sequencialmente, conforme descrito por Von Neumann (1945). Sendo assim, o tempo levado por uma computação é o mesmo necessário para que todas as instruções sejam executadas pelo processador. Como elas são executadas em sequência, o tempo total da computação é a soma dos tempos de cada instrução.

Uma forma de agilizar a execução de um programa computacional (conjunto de instruções) é a utilização do conceito e processamento paralelo, cuja ideia básica é a utilização de várias unidades de processamento, ao mesmo tempo. O programa, originalmente sequencial, é particionado de forma que partes independentes possam ser executadas ao mesmo tempo em diferentes unidades de processamento.

Para dar suporte ao processamento paralelo, foram concebidas arquiteturas computacionais especiais, com diversas unidades de processamento, conhecidas como supercomputadores ou MPPs (*Massively Parallel Processors*) e os *clusters*.

Os MPPs são imensos computadores que custam muitos milhões de dólares. Inicialmente, eles eram usados em ciências, na engenharia e na indústria para cálculos muito grandes, também para tratar de um elevado número de transações por segundo, e até para armazenamento e gerenciamento de imensos bancos de dados. Devido ao alto custo dos MPPs, a utilização de aglomerados de computadores, chamados de *clusters*, passou a ser uma tendência mundial.

Computação em *cluster* também pode ser usada como uma forma de relativo baixo custo de processamento paralelo para aplicações científicas e outras que se prestam a operações desta natureza. Um dos primeiros exemplos, bastante conhecido, foi o projeto *Beowulf*,[1] em que um número de *off-the-shelf* PCs foi usado para formar um *cluster* para aplicações científicas.[2]

O que faz com que os *clusters* empurrem os MPPs para nichos cada vez menores é o fato de que suas atuais interconexões de alta velocidade são encontradas com facilidade no mercado. Fenômeno semelhante aconteceu quando os PCs transformaram os *mainframes* em raridades nos dias atuais (TANENBAUM, 2001, p. 367).

O desenvolvimento de programas em *clusters* necessita de *softwares* específicos, usualmente bibliotecas, que manipulem a comunicação e a sincronização entre processos. Estes *softwares* são conhecidos como sistemas de troca de mensagens. Freitas,[3] em relação à necessidade do uso de *softwares* específicos de *clusters*, diz:

> Uma rede de computadores pode ser considerada como uma excelente opção para a execução dos problemas de MPPs [e *cluster*]; logo, buscou-se implementar e viabilizar, através de um protocolo baseado na troca de

[1] Com a finalidade de atender às necessidades de processamento do Goddard Space Flight Center da NASA, no verão de 1994, Thomas Sterlinge e Don Becker construíram um computador paralelo inteiramente incomum. Possuía 16 processadores DX4 da Intel conectados por rede Ethernet de 10Mbit/s e *softwares* disponíveis gratuitamente, a começar pelo sistema operacional Linux, compiladores GNU, e suportando programação paralela com MPI (mensagem passagem), tudo disponível gratuitamente (QUINN, Michael J. **Parallel Programming in C with MPI and OpenMP**. Oregon: Oregon State University, 2004. p. 12).

[2] Disponível em: http://searchdatacenter.techtarget.com/definition/cluster-computing. Acessado em: 31 jan. 2012.

[3] Disponível em: http://www.inf.ufrgs.br/gppd/disc/cmp134/trabs/T2/981/mpi.html. Acessado em: 31 jan. 2012.

> mensagens, o uso de ferramentas que utilizassem os diversos processadores existentes em uma rede, bem como suas respectivas memórias locais. Assim, surgiram, como alternativas a esta forma de execução de aplicações, os ambientes MPI e PVM.

O PVM (*Parallel Virtual Machine*) é uma biblioteca que nasceu nos laboratórios da *Emory University* e do *Oak Ridge National Laboratory*, e permite que uma rede heterogênea de computadores de todos os tipos se comporte como sendo apenas uma única máquina paralela virtual expansível e concorrente.

MPI, acrônimo para *Message Passing Interface*, é o resultado de um padrão especificado por um Fórum aberto formado por pesquisadores, empresas e usuários que definiram a sintaxe, a semântica e o conjunto de rotinas. Embora de origens bastante diferentes, PVM e MPI possuem especificações para as bibliotecas que podem ser usadas em computação paralela. Com componentes comuns de rotinas de processos como funções para inicializar, finalizar, determinar e identificar processos, rotinas ponto a ponto e de *broadcast*, tanto MPI como PVM são muito semelhantes, sendo que o MPI é mais fácil de usar e possui uma interface de comunicação mais confiável, descartando do desenvolvedor preocupações com possíveis falhas de comunicação.

Motivados pela demanda de alto custo computacional exigida por aplicações gráficas, surgiram nos anos 1980 os aceleradores gráficos não programáveis que, em meados dos anos 1990, passaram a integrar todos os elementos essenciais em um único *chip*. Esta integração incorporou funcionalidades como *pipeline* 3D completa, um ganho expressivo na largura de banda da memória dos *chips* em relação à CPU; consequentemente, nasce a computação nas unidades de processamento gráfico (GPUs).[4]

O desempenho do processamento em GPU tornou-se tão expressivo que despertou a atenção de pesquisadores e programadores de outras áreas (que não a computação gráfica), que passaram a aprimorar formas de utilização das GPUs para propósitos gerais (como a computação científica, por exemplo). A técnica era chamada GPGPU – “programação de uso geral usando uma unidade de processamento gráfico”.

Até o lançamento do GeForce 8800 GTX, ou simplesmente G80, no final de 2006, pela NVIDIA, eram necessárias técnicas de OpenGL ou Direct3D para

[4] NVIDIA’s Fermi: The First Complete GPU Computing Architecture.

programá-lo. Fazer isto, na época, não era tarefa fácil (KIRK, 2011, p. 4-6). Com a introdução do modelo de programação CUDA (NVIDIA, 2007), acrônimo para *Compute Unified Device Architecture*, passou-se a admitir a execução ora CPU ora GPU (programação híbrida) de uma aplicação e a usar as ferramentas de programação C/C++ tradicionais; tudo isto graças ao *hardware* adicional acrescentado ao G80 e seus *chips* sucessores para facilitar a comodidade da programação paralela (KIRK, 2011, p. 4-6). Com isto, os processadores G80 e seus sucessores venderam mais de 200 milhões de unidades, tornando viável, pela primeira vez, a computação maciçamente paralela com um produto do mercado em massa (KIRK, 2011, p. 5).

Nesse cenário, as GPUs, originalmente concebidas para aplicações gráficas desde o princípio, possuem suporte para o processamento paralelo, tornando-se, hoje, dispositivos indispensáveis para o desenvolvimento de computação de alto desempenho, seja nas pesquisas no meio acadêmico ou nas indústrias. Três entre os cinco computadores mais rápidos em nível mundial utilizam GPUs NVIDIA, segundo o *ranking* Top 500*,* publicado pela International Supercomputing Conference e pelo ACM/IEEE Supercomputing Conference, em junho e novembro de 2011, respectivamente.[5]

Buscando encontrar melhorias nas soluções que demandem alto custo computacional, este trabalho pretende, em continuidade à pesquisa desenvolvida por Almeida (2009), compreender e explorar a capacidade da alta *performance* de uma arquitetura computacional composta por um aglomerado (*cluster*) de máquinas com GPUs. Nesta arquitetura híbrida são explorados tanto os conceitos de programação em *clusters* de computadores (por meio de MPI) quanto a programação de GPU (por meio de CUDA).

No presente trabalho, são desenvolvidos programas utilizando-se linguagem C, MPI e CUDA, e o Método Monte Carlo para resolver o problema do transporte de nêutrons, empregado no cálculo de uma blindagem. Primeiramente, usa-se uma solução sequencial; posteriormente, soluções paralelas em 1 GPU, 2 GPUs e, finalmente, o *cluster* de GPUs com a finalidade de compreender e demonstrar como um *cluster* de GPUs pode contribuir com ganhos de *performance* na solução de problemas típico da Engenharia Nuclear que demandam alto custo computacional.

[5] A GPU NVIDIA Tesla é usada no supercomputador mais rápido do mundo, o Tianhe-1A da China – Fonte: http://www.nvidia.com.br/page/corporate_timeline.html. Acessado em: 20 dez. 2011.

Dessa forma, pretende-se apresentar soluções para o referido problema, fazendo-se uso de um *cluster* formado por quatro microcomputadores com processador i7-960, 2 GPUs GTX 480 Fermi com 480 núcleos da NVIDIA (cada uma), executando num sistema operacional Linux, distribuição Fedora 16 e MPI para comunicação entre os microcomputadores.

Este trabalho encontra-se organizado da seguinte forma:

No Capítulo 2, aborda-se a utilização da computação paralela, com o objetivo de se reduzir o tempo necessário à resolução de um único problema computacional.

O Capítulo 3 trata da simulação do transporte de nêutrons através de uma blindagem conhecida como *slab*, utilizando-se o Método Monte Carlo.

O desenvolvimento de diversas implementações das soluções, visando-se avaliar o ganho de *performance* de cada implementação, é tratado no Capítulo 4.

O Capítulo 5 trata dos experimentos feitos por meio da simulação do problema do transporte de nêutrons através de *slab*, usando-se o Método Monte Carlo, e avaliando-se os respectivos resultados.

Finalmente, o Capítulo 6 apresenta conclusões do presente trabalho, bem como indica possíveis desdobramento do mesmo.

2 COMPUTAÇÃO PARALELA

2.1 Aspectos gerais

Comparativamente, os atuais computadores são cerca de 100 vezes mais rápidos do que há apenas uma década, mas, dependendo de quais forem as aplicações, este “mais rápido” não é o suficiente. Pesquisadores fazem uma série de simplificações para os problemas e ainda têm de esperar horas, dias ou mesmo semanas para que os programas possam concluir a execução. Se um problema de alto custo computacional leva 12 horas para produzir um resultado e, de repente, ele roda 10 vezes mais rápido, poderá produzir, então, uma resposta em menos de duas horas (QUINN, 2004).

Simplesmente poderia esperar-se por CPUs mais rápidas. Daqui a cinco anos, CPUs serão 10 vezes mais rápidas do que hoje. Ao longo das últimas décadas, o crescimento do poder computacional dos processadores tem obedecido à Lei de Moore (QUINN, 2004), que afirma que a capacidade dos processadores dobra a cada 18 meses,[6] porém, segundo alguns pesquisadores, a Lei de Moore está chegando ao seu fim, seja por novas tecnologias que surgirão – mais rápidas, sofisticadas e econômicas – ou pode, até mesmo, ser ampliada devido à diminuição do consumo de energia pelas baterias e aumento de desempenho (GEIST *et al.* 1994).

A computação paralela é uma forma comprovada para se obter maior desempenho agora (QUINN, 2004). Ela é a utilização de um computador paralelo visando reduzir o tempo necessário para resolver um único problema computacional. Atualmente, a computação paralela é considerada uma forma padrão para pesquisadores resolverem problemas em diversas áreas, tais como evolução galáctica, modelagem climática, *design* de aeronaves, dinâmica molecular e, especificamente, problemas na área nuclear, como recarga de reatores (WAINTRAUB *et al.*, 2009) , projetos neutrônicos (PEREIRA; LAPA; MOL, v. 4, p. 416-420, 2002) e termo-hidráulico (PEREIRA, v. 30, p. 1.665-1.675, 2003), e tantos outros que hão de vir.

---

[6] Disponível em: http://olhardigital.uol.com.br/produtos/digital_news/noticias/a-lei-de-moore-esta-chegando-ao-fim. Acessado em: 29 dez. 2011.

2.1.1 Classificação de computação – taxonomia de Flynn

Vários tipos de computadores paralelos já foram propostos e construídos até hoje. Ao mesmo tempo, muitos pesquisadores tentaram classificá-los em uma taxonomia (TANENBAUM, 2001), sendo que Michael Flynn foi quem idealizou e divulgou uma das mais conhecidas classificações de computadores paralelos (FLYNN, 1972). Sua classificação é apresentada pela Tabela 2.1 e se refere à maneira como instruções e dados são organizados em um determinado tipo de processamento. Todos os tipos definidos na taxonomia de Flynn mencionam o elemento "fluxo de dados", que significa um conjunto de dados, e "fluxo de instruções", que indica um conjunto de instruções a serem executadas.

Tabela 2-1 – Taxonomia de Flynn para computadores paralelos

	Single Instruction	Multiple Instruction
Single Data	**S I S D** (Single Instruction, Single Data)	**M I S D** (Multiple Instruction, Single Data)
Multiple Data	**S I M D** (Single Instruction, Multiple Data)	**M I M D** (Multiple Instruction, Multiple Data)

Os quatro tipos de processamento são:

● SISD – *Single Instruction Single Data* – considera-se um único conjunto de instruções e de dados. Enquadram-se nesta classificação as máquinas cuja arquitetura segue o padrão Von Newmann, isto é, processadores que executam uma instrução completa de cada vez, sequencialmente, cada uma delas manipulando um dado específico ou os dados daquela operação. Como exemplo, citamos o processamento escalar.

● MISD – *Multiple Instruction Single Data* – trata-se de um tipo de arquitetura que pode usar múltiplas instruções para manipular apenas um conjunto único de

dados, como um vetor. Um exemplo desta categoria de arquitetura é o do processador vetorial.

● SIMD – *Single Instruction Multiple Data* – neste tipo de arquitetura, o processador opera de modo que uma única instrução acessa e manipula um conjunto de dados, simultaneamente. Neste caso, a unidade de controle do processador aciona diversas unidades de processamento.

● MIMD – *Multiple Instruction Multiple Data* – é a categoria mais avançada tecnologicamente, produzindo elevado desempenho do sistema de computação. A categoria MIMD refere-se a computadores com múltiplos fluxos de instrução e múltiplos fluxos de dados. Os multiprocessadores e os multicomputadores encaixam-se nesta categoria, sendo que ambas as arquiteturas baseiam-se em processadores múltiplos. Assim, CPUs diferentes podem executar simultaneamente diferentes fluxos de instrução e manipular fluxos de dados diferentes.

2.2 GPU – *Graphics Processing Unit*

Segundo Hwu *et al.* (2008), desde 2003, a indústria de semicondutores tem estabelecido duas trajetórias principais para o projeto de microprocessadores. A trajetória de múltiplos núcleos (*multicore*) começou com processadores de dois núcleos. Um exemplo atual é o microprocessador Intel Core i7[7] 3960X com seis núcleos processadores, cada um sendo um processador independente, com o microprocessador admite *hyperthreading* com dois *threads* de *hardware* e foi projetado para maximizar a velocidade de execução dos programas sequenciais. A Figura 2-1 mostra a porção da área *die*[8] usada por ALUs no processador Core i7 (codinome Sandy Bridge), da Intel.

[7] Sobre i7, confira-se o *site* http://ark.intel.com/pt-br/products/63696/Intel-Core-i7-3960X-Processor-Extreme-Edition-(15M-Cache-up-to-3_90-GHz).

[8] Circuitos integrados são produzidos em grandes lotes num único *wafer* de EGS (*Electronic Grade Silicon*), através de processos tais como litografia. O *wafer* é cortado em muitos pedaços, cada um contendo uma cópia do circuito. Cada um destes pedaços é chamado de *die*. Disponível em: http://en.wikipedia.org/wiki/Die_(integrated_circuit). Acessado em: 20 abr. 2012.

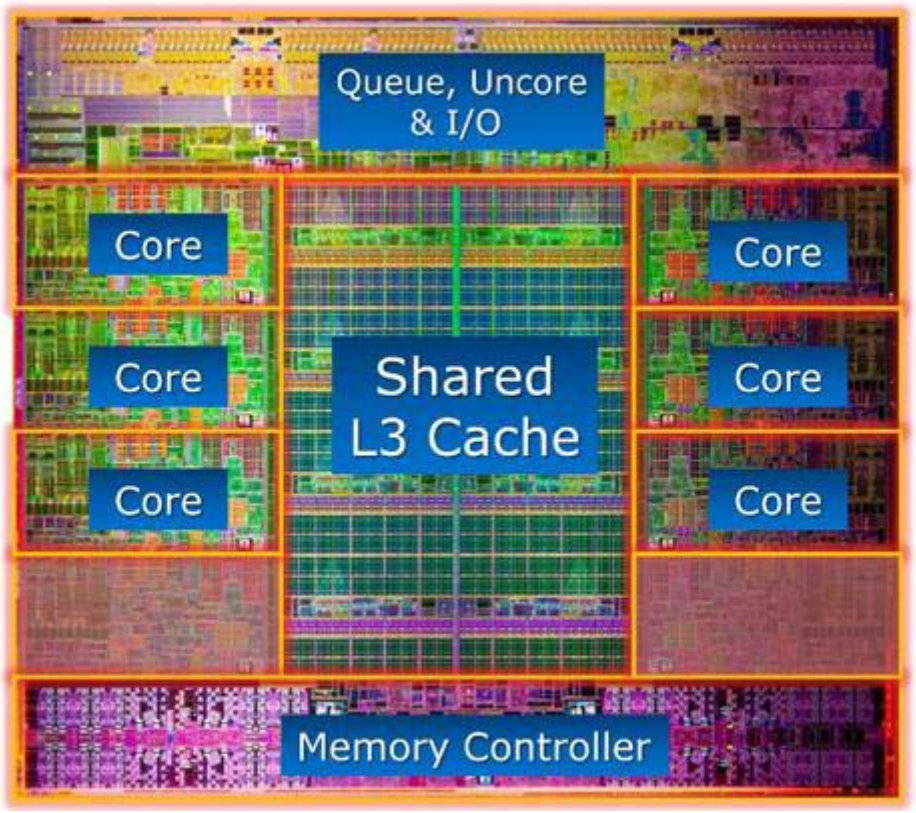

Figura 2-1 – Processador Intel Core i7 3960X com seis núcleos de CPU, 15MB de cache L3, e controladores *on-chip* DRAM.
Fonte: Intel Corporation, exceto destaques

Por outro lado, a trajetória baseada em muitos núcleos (*many-core*), como as GPUs – *Graphics Processing Unit*, concebidas desde 2002, originariamente para aplicações gráficas, foca-se em execução de várias aplicações paralelas. Um exemplo atual é a GTX 480, lançada em abril de 2010,[9] com 480 núcleos e tecnologia Fermi.[10] No Gráfico 2-1, observa-se a diferença de pico de desempenho entre CPUs e GPUs.

Um processador Intel atinge apenas alguns GFlops/s, enquanto 1 GPU GTX 480, pode atingir mais que 1.25 TeraFlop/s. Também pode-se observar no Gráfico 2-2 que, a partir de novembro de 2006, as GPUs da NVIDIA estabeleceram uma larga vantagem no poder computacional em relação às CPUs da Intel.

[9] Disponível em: http://www.geforce.com/Hardware/GPUs/geforce-gtx-480/specifications. Acessado em: 29 fev. 2012.
[10] Disponível em: http://www.nvidia.com.br/object/prbr_033110.html. Acessado em: 29 fev. 2012.

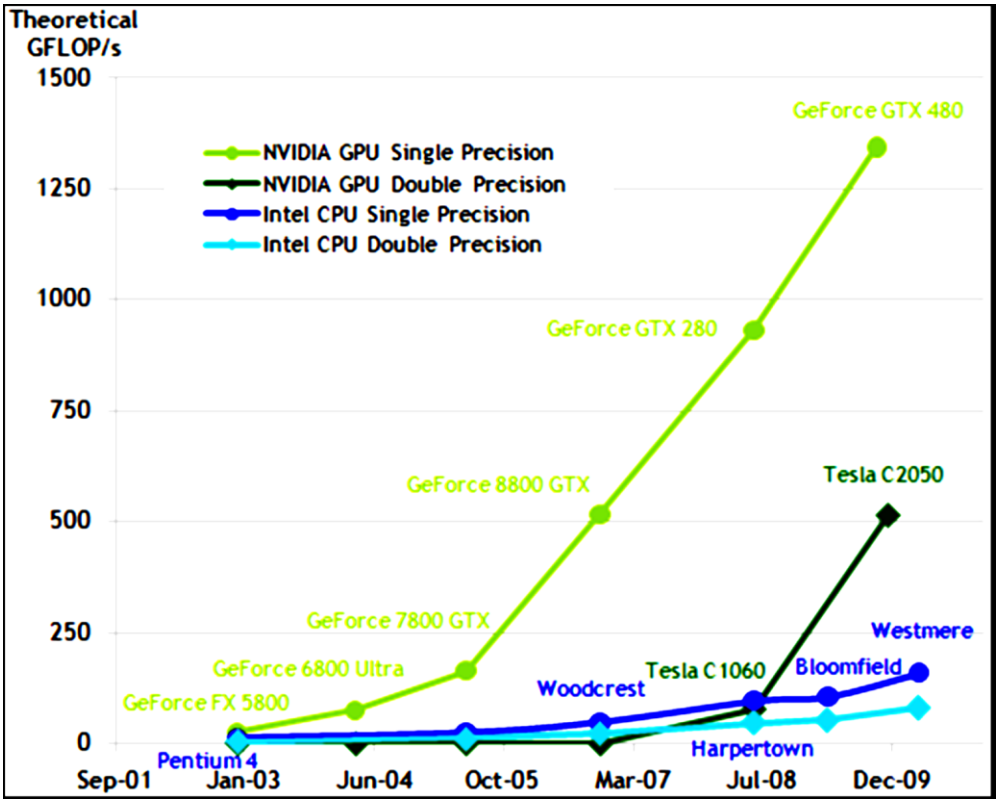

Gráfico 2-1 – Comparação GPU x CPU
Fonte: NVIDIA CUDA C Programming Guide

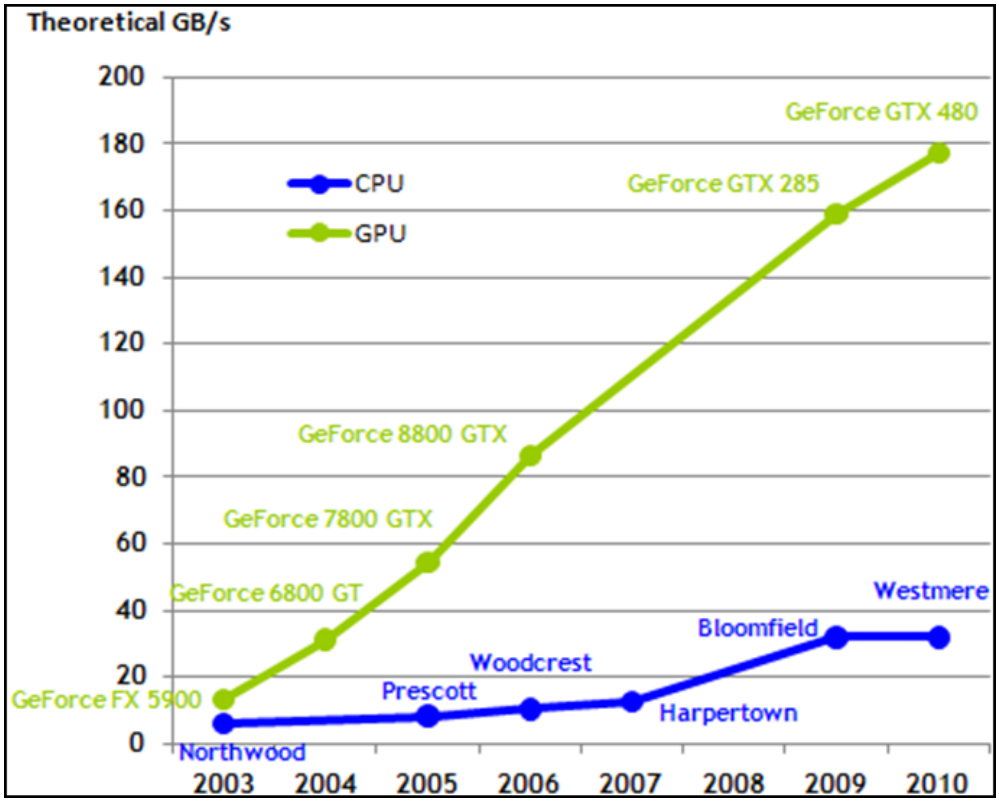

Gráfico 2-2 – Operações de ponto flutuante por segundo e largura de banda de memória para a CPU e a GPU
Fonte: NVIDIA CUDA C Programming Guide

A razão para a lacuna de desempenho ser imensa entre CPU com múltiplos núcleos e GPU com muitos núcleos está na filosofia de projetos fundamentais nos dois tipos de processadores, como esquematicamente ilustrado pela Figura 2-2.

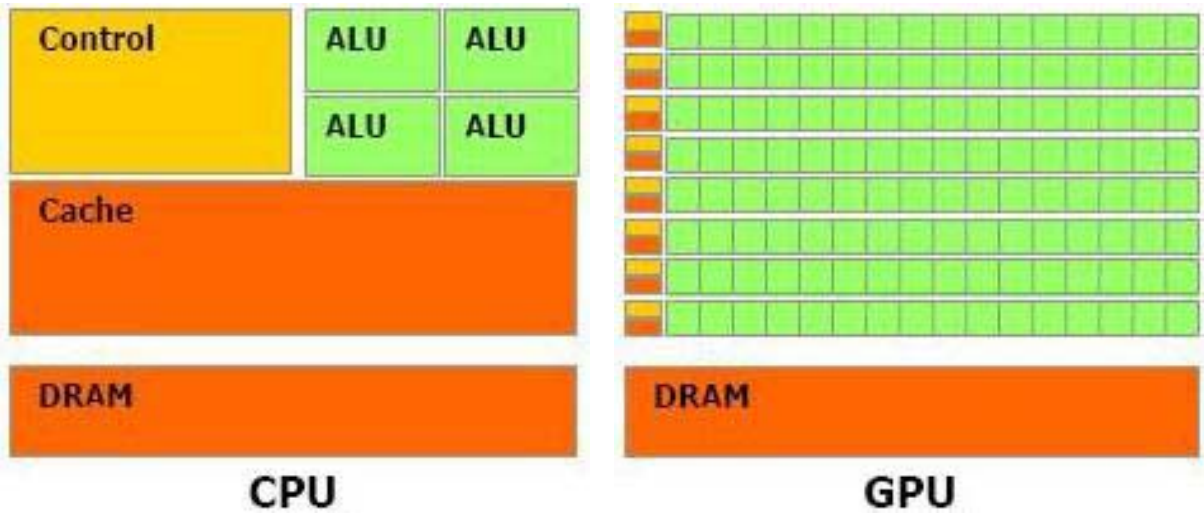

Figura 2-2 – Filosofia de projetos diferentes entre CPUs e GPUs. Na GPU são dedicados mais transistores de Processamento de Dados
Fonte: NVIDIA CUDA C Programming Guide

Para redução da latência de acesso a instruções e dados de grandes e complexas aplicações, nos projetos dos microprocessadores de muitos *cores*, os mesmos são fornecidos com grande memória *cache* e fazem uso de uma lógica de controle que permite que instruções de um único *thread* de execução sejam executadas individualmente pelos *cores* fora de sua ordem sequencial, ou seja, em paralelo (KIRK, 2011, p. 3-4). A GPU é especializada para o tipo de computação paralela intensiva e, portanto, projetada de tal forma que mais transistores são dedicados ao processamento de dados em vez de *cache* de dados e controle de fluxo.

As GPUs foram concebidas originariamente para processar grande conjunto de dados, por exemplo, na renderização gráfica, que são dados em posições **contínuas** na memória, caracterizando um vetor ou *array* de dados. Por outro lado, no processamento de dados que não envolva gráficos, as operações requisitadas não se realizam predominantemente sobre posições contínuas da memória. A cada ciclo de *clock*, as operações são realizadas sobre posições diferentes do *array,* que neste caso*,* pode estar armazenado em posições **não contínuas** da memória. Este tipo de processamento é chamado de escalar, e os processadores para este fim, diz-

se “processador escalar”, como as tradicionais CPUs. Na taxonomia de Flynn diz-se que são SISD.

Veja-se o caso de um processador escalar que tenha de realizar a soma de dois vetores A e B, com mil posições cada, e gerar um terceiro vetor C. Nosso processador escalar terá de apontar para as primeiras posições de A e B, copiar os valores ali armazenados para os registradores da CPU, efetuar a soma e transferir o resultado para uma posição de memória destinada ao vetor C. Em seguida, incrementar os ponteiros de memória e repetir toda a operação até somar as mil posições dos vetores A e B.

Isso torna os processadores escalares ineficientes para esse tipo de operação, devido ao fato de o tempo gasto para gerenciar os ponteiros executar a contagem dos ciclos e demais operações auxiliares costumar ser muito maior do que o gasto para executar o processamento propriamente dito, ou seja, as mil operações de soma dos elementos do *array*. Tipicamente, é desta maneira que um processador escalar trabalha. Já uma GPU funciona de forma diferente. Ela se destina principalmente a processar gráficos que se caracterizam por um enorme número de dados similares em posições contínuas da memória principal, e a imensa maioria das operações a serem por ela executadas tem como objeto os *arrays*. Por isso, as GPUs adotam um projeto na sua arquitetura, que otimiza o processamento dos elementos de um grande *array.*

Consequentemente, esse tipo de processador se distingue por sua grande capacidade de executar cálculos simultâneos sobre um conjunto de dados. Pela taxonomia de Flynn, as GPUs são SIMD e, por serem concebidas para operarem principalmente sobre *arrays*, elas são classificadas como “processadores vetoriais”.

Numa GPU preparada para rodar CUDA existe um número de processadores chamados *streaming* (SPs), que compartilham a lógica de controle e a *cache* de instruções, e são capazes de operar simultaneamente. Os processadores *streaming* ficam organizados em um *array*, chamado de multiprocessador de *streaming* (SM).

O estado da arte em *design* de GPU é representado pela nova geração da NVIDIA CUDA, arquitetura de codinome Fermi. A Figura 2-3 mostra um diagrama de blocos de alto nível do primeiro *chip* Fermi, e a Figura 2-4 mostra, em detalhe, um bloco de SPs.

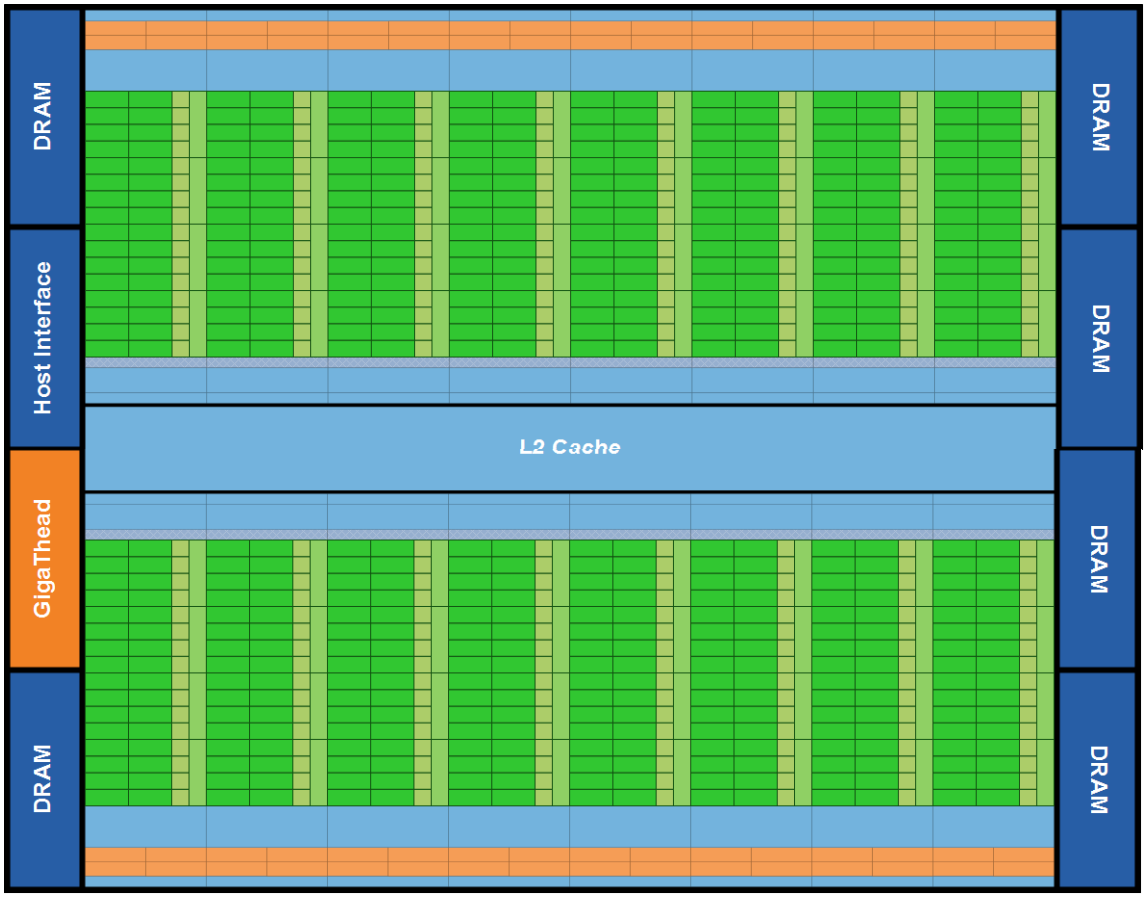

Figura 2-3 – GPU NVIDIA arquitetura Fermi consiste em vários multiprocessadores *streaming* (SMs).
Fonte: NVIDIA

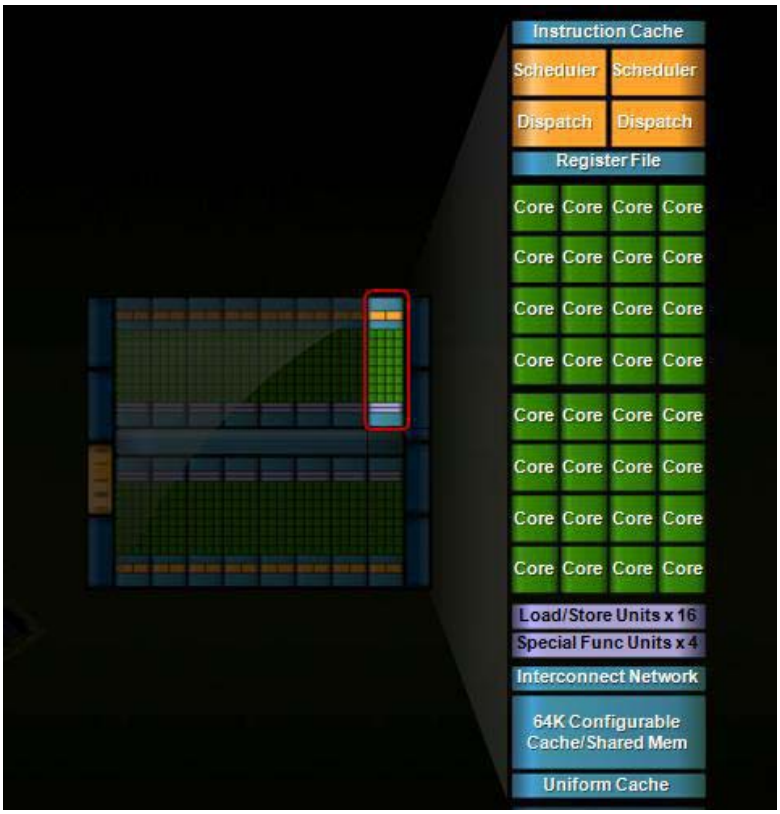

Figura 2-4 – Detalhe de um multiprocessador *streaming* (SM). Cada SM é constituído por 32 núcleos, cada um dos quais pode executar uma instrução de ponto flutuante ou inteiro por *clock*.
Fonte: NVIDIA

No exemplo de somar dois *arrays* A e B, com mil posições cada, e supondo que a GPU possui 100 SPs, para efetuar a soma, as mil operações são distribuídas pelas 100 unidades SPs, e todo o cálculo é realizado em um décimo do tempo gasto para se efetuar a mesma operação usando uma CPU.

Outra questão relevante é a largura de banda da memória. Os *chips* da GPU operam em aproximadamente 10 vezes a largura de banda dos *chips* de CPU disponíveis na mesma época. Em 2010, o GeForce GTX 480 foi capaz de mover dados em cerca de 180 GB/s para dentro e para fora de sua memória de acesso aleatório dinâmica (DRAM) principal. Outros fatores podem limitar os valores de largura de banda. Como vários *softwares* de sistemas, aplicações e *devices* de entrada e saída esperam que seus acessos à memória funcionem, os processadores de uso geral precisam satisfazer os requisitos dos sistemas operacionais, aplicações e *devices* de E/S, o que torna a largura de banda da memória mais difícil de aumentar.

Embora originalmente concebidas para o processamento intensivo exigido na computação gráfica, as GPUs são coprocessadores genéricos, ou seja, dispositivos

capazes de realizar instruções computacionais de forma altamente paralela. Tal observação deu início a uma nova abordagem, a GPU Computing (também chamada GPGPU – *General-Purpose Computation on GPU*), conceito que visa explorar as vantagens das placas gráficas modernas com aplicações de propósito geral altamente paralelizáveis que exigem intenso fluxo de cálculos. Com seu crescente potencial no futuro dos sistemas computacionais, várias aplicações com particularidades semelhantes têm sido identificadas e executadas com sucesso nas GPUs.

2.3 CUDA

Com o advento do novo paradigma introduzido pelo *hardware* gráfico programável, as GPUs, várias soluções têm sido propostas para facilitar o processo de acesso a estas unidades programáveis:

> ... computação de propósito geral em unidades de processamento gráfico (GPGPU) está empurrando o que antes era um dispositivo gráfico dedicado tornando-o dispositivo para executar tarefas tradicionalmente tratados pela CPU. CUDA, StreamSDK, RapidMind e OpenCL representam hoje a interface de programação mais recente para o *hardware* de computação paralela como GPGPUs. (BORGO; BRODLIE, 2009, p. 13).

Em novembro de 2006, a NVIDIA lança a CUDA™.[11] Acrônimo para Compute Unified Device Architecture, a CUDA é uma arquitetura de *hardware* e *software* que permite explorar toda capacidade das GPUs NVIDIA de executar programas em paralelo e computação de alto desempenho escritos em C, C++, Fortran, OpenCL, DirectCompute e outras linguagens.[12] Presente a partir da série GeForce 8800 (G80), e em busca de outros mercados como computação de alto desempenho (HPC), a NVIDIA lança a linha Tesla, baseada na arquitetura GT200 (GLASKOWSKY, 2009, p.15). Esta estratégia contribuiu para que se tenha uma

[11] NVIDIA CUDA C Programming Guide, Version 3.2, 22/10/2010.
[12] Disponível em: http://www.nvidia.com/object/cuda_home_new.html. Acessado em: 02 jan. 2012. White Paper NVIDIA's Next Generation CUDA Compute Architecture: Fermi. NVIDIA Corporation, 2009.

significativa base instalada de desenvolvedores de aplicativos.[13] Os elevados volumes de vendas de GPUs, embora impulsionados principalmente pelos mercados de *games,* fazem, hoje, as GPUs mais atraentes para os desenvolvedores de aplicações de HPC do que para os dedicados a supercomputadores de empresas como, a Cray, a Fujitsu, a IBM, a NEC e a SGI (GLASKOWSKY, 2009, p. 15). A NVIDIA diz que já entregou mais de 100 milhões de CUDA para PCs (HALFHILL, 2009), isto é, mais de 100 milhões de PCs têm GPUs capazes de executar CUDA.

2.3.1 Arquitetura CUDA

A primeira geração de GPU, G80, possuía 128 *shaders* programáveis, mais tarde chamados de núcleos de processador de *streaming* (SP), e agora conhecidos simplesmente como núcleos CUDA. Oito núcleos CUDA foram agrupados em um multiprocessador *streaming* (SM), cujos núcleos compartilham recursos comuns, como memória local, registros de arquivos, unidades de *load*/*store*, escalonadores de *threads*. A G80 tinha 16 destes multiprocessadores *streaming* (16 SMs x 8 SPs por cada SM, totalizando 128 núcleos por *chip*). A Figura 2-5 mostra a arquitetura de uma GPU G80 onde se pode observar os oito SMs e seus 16 SPs (HALFHILL, 2009, p. 4).

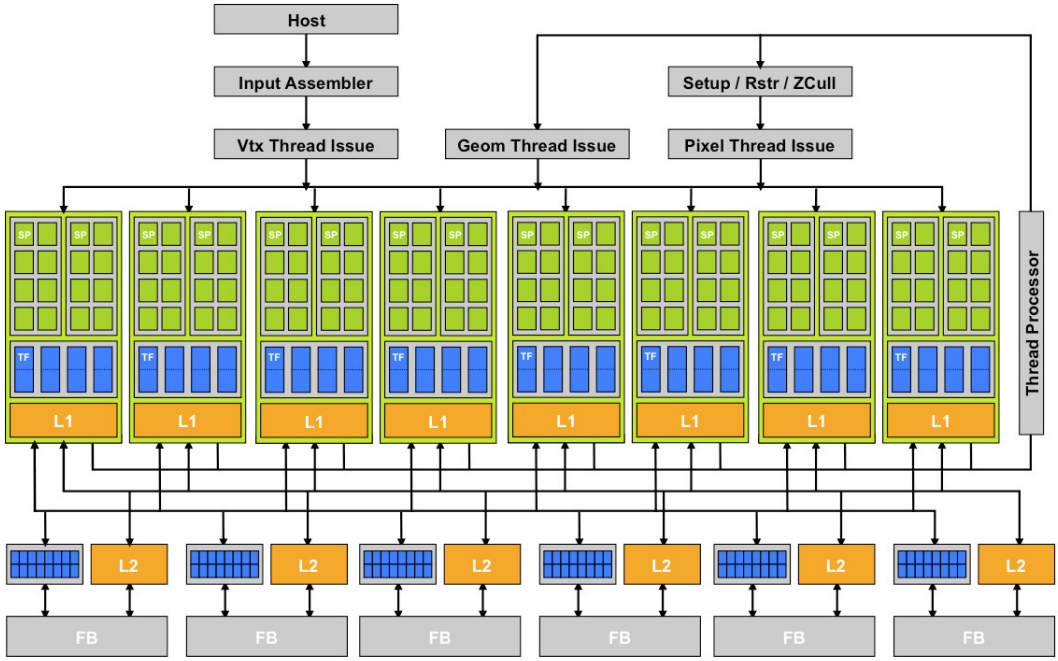

Figura 2-5 – Arquitetura da GPU GeForce 8800

[13] Disponível em: http://www.developer.nvidia.com/what-cuda. Acessado em: 02 jan. 2012.

A G80 foi a visão inicial de algo maior. A unificação do processamento gráfico com a computação paralela originou a arquitetura GT200; lançada em junho de 2008, estendeu o desempenho e a funcionalidade do G80. Entretanto, Fermi, pode-se dizer, a 3ª geração, foi o resultado de todo o aprendizado anterior, que empregou uma abordagem completamente nova para projetar e criar o primeiro GPU computacional.[14] Fermi substituiu a arquitetura GT200. Tem 32 núcleos CUDA por multiprocessador *streaming* – quatro vezes a GT200 e G80. GPUs Fermi possuem 16 multiprocessadores de *streaming*, gerando um total até 512 núcleos CUDA por *chip*.

A Figura 2-6 ilustra uma arquitetura Fermi. No detalhe, vê-se um único núcleo CUDA e sua relação com o fluxo de multiprocessador, ao qual ele pertence. Apesar de um núcleo CUDA se assemelhar a uma CPU *multicore* x86, ele é muito mais simples, mas tem toda a capacidade de um *pixel shader.* Cada núcleo CUDA tem uma unidade de ponto flutuante (FP Unit, padrão IEEE 754-2008), uma unidade de inteiro (INT Unit, 64 bit), alguma lógica para o envio de instruções e operandos para estas unidades, e uma fila para a realização de resultados (HALFHILL, 2009, p. 6). Os multiprocessadores *streamings* compreendem 32 núcleos CUDA, cada um dos quais podendo executar operações de ponto flutuante e inteiro, junto com 16 unidades *load-store* para operações de memória, quatro unidades de funções especiais (SFUs) e 64 KB de RAM com particionamento configurável de memória compartilhada e cache L1 (GLASKOWSKY, 2009, p. 19).[15] As unidades de função especial estão disponíveis para lidar com operações transcendentais, seno, cosseno, exponencial e outras.[16] Estes recursos são compartilhados entre todos os 32 núcleos CUDA em um multiprocessador *streaming*.

[14] White Paper NVIDIA's Next Generation CUDA Compute Architecture: Fermi. NVIDIA Corporation, 2009. p. 4.
[15] White Paper NVIDIA's Next Generation CUDA Compute Architecture: Fermi. NVIDIA Corporation, 2009. p. 5
[16] White Paper NVIDIA's Next Generation CUDA Compute Architecture: Fermi. NVIDIA Corporation, 2009. p. 9.

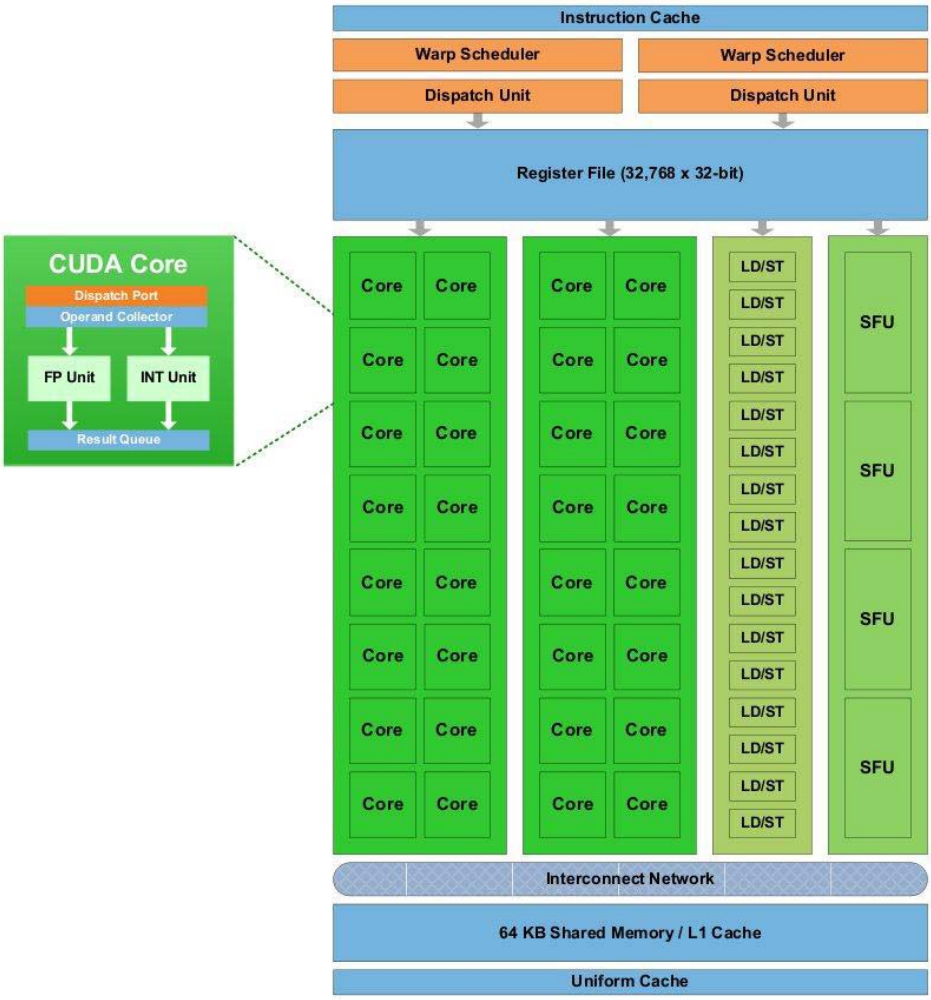

Figura 2-6 – Multiprocessador *streaming* Fermi com seus 32 núcleos CUDA

Esses 32 núcleos são projetados para trabalhar em paralelo com 32 instruções por vez, a partir de um conjunto de 32 *threads*, que a NVIDIA chama de *warp* (HALFHILL, 2009, p. 7).

Dentro do SM, os núcleos estão divididos em dois blocos de execução com 16 núcleos cada. Cada bloco possui duas unidades desacopladas e independentes chamadas *warp scheduler* e *instruction dispatch unit*. Em cada ciclo, um total de 32 instruções pode ser entregue a partir de um ou dois *warps* a estes blocos. Um *warp* é a unidade de escalonamento de *threads* por SMs (KIRK, 2011, p. 59). Dois ciclos são necessários para as 32 instruções de cada *warp* serem executadas nos núcleos. Estas unidades alternam os *threads* e as *warps* para manter os núcleos CUDA o máximo de tempo ocupados. A GPU pode mudar de um *thread* para outro em cada ciclo de *clock*.

A Figura 2-7 mostra uma sequência de instruções a serem distribuídas entre os blocos de execução disponíveis (GLASKOWSKY, 2009). Cada multiprocessador de *streaming* pode gerenciar 48 *warps*. Como cada *warp* tem 32 *threads*, um multiprocessador de *streaming* pode gerenciar 1.536 *threads*. Com 16 SMs, uma GPU Fermi pode manipular até 24.576 *threads* paralelos (HALFHILL, 2009).

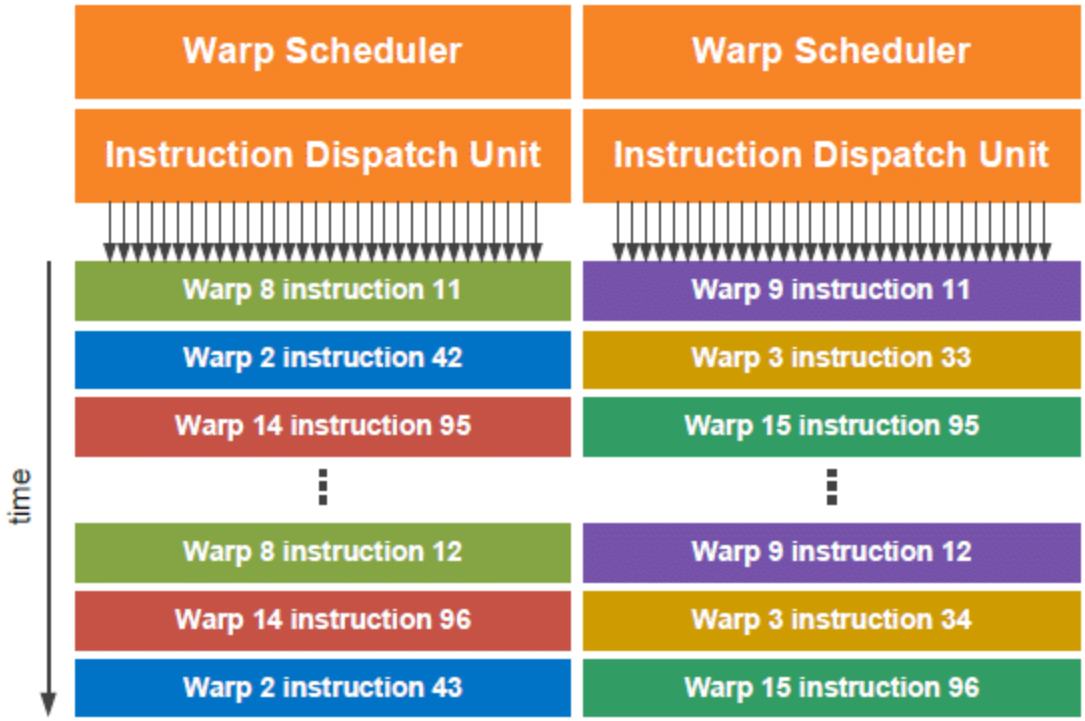

Figura 2-7 – Escalonamento de *warps* por ciclo de instrução em um SM Fermi

Para alcançar *multithreading* massivamente paralelo, os processadores CUDA precisam executar eficientemente operações de alta latência, como os acessos à memória global ou aritmética de ponto flutuante. Se uma determinada instrução executada pelos *threads* em um *warp* precisa ficar aguardando pelo resultado de uma operação com alta latência, um mecanismo de prioridade seleciona outra instrução do outro *warp* residente que está pronto para ser executado. Geralmente chama-se esta operação de ocultação de latência. Esta capacidade das GPUs é fundamentalmente fruto do projeto que não dedica grandes quantidades de memória *cachê*, como fazem as CPUs.

Cabe-nos, nesse momento, falar um pouco da GPU GTX480, *hardware* usado neste trabalho. Baseada na tecnologia Fermi, a GTX480 possui 32 núcleos CUDA por multiprocessador *streaming* – quatro vezes o que possui uma GT200 e uma

G80. Ela possui 15 multiprocessadores de *streaming*, para um total de 480 núcleos CUDA por *chip*. Os principais recursos em cada GPU estão disponíveis através de um padrão chamado *compute capability*. Especificamente, a GTX480 possui *compute capability* 2.0, e todas as suas especificações técnicas e recursos encontram-se no manual de programação CUDA C Programming Guide Version 3.2, no Apêndice G. Estas especificações podem ser encontradas na Figura 2-8.

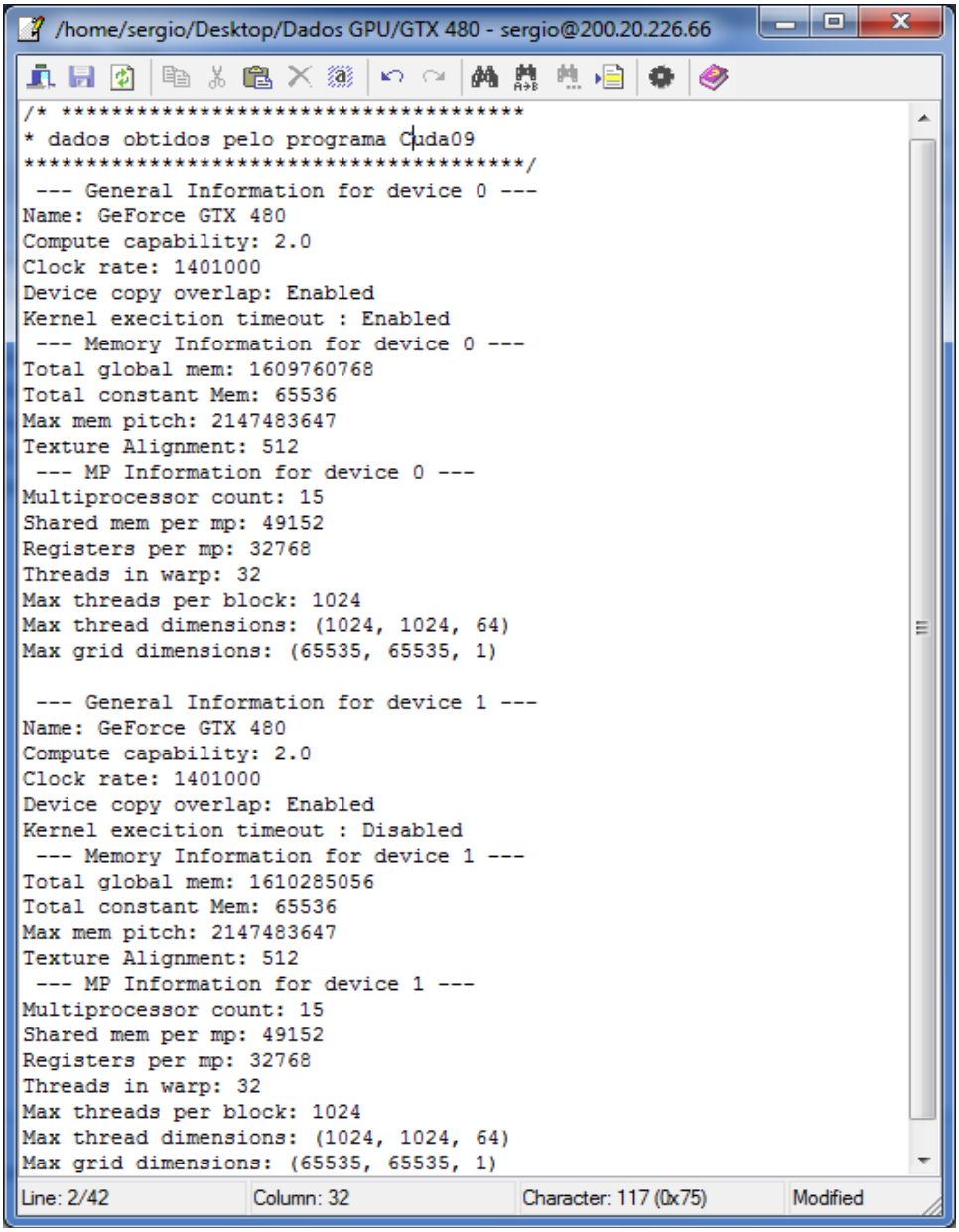

Figura 2-8 – *Overview* da GTX 480 gerado por um programa CUDA

2.3.2 Programação em CUDA

A NVIDIA não poupou esforços para facilitar a maneira de programar a complexa arquitetura Fermi, permitindo aos desenvolvedores de *software* concentrarem-se no *design* algoritmo, em vez de detalhes de como mapear o algoritmo para o *hardware*, melhorando assim a produtividade (KIRK, 2011, p. 6; GLASKOWSKY, 2009, p. 17).

Um programa CUDA é constituído por duas partes: um código escrito em ANSI C e um código em ANSI C Extended. O código escrito em ANSI C é executado pela CPU (*host*), e o ANSI C estendido permite escrever funções chamadas de *kernels e* é executado em um ou mais dispositivos GPUs (*devices*). *Kernels* possuem palavras-chave adicionais para expressar o paralelismo diretamente, e não através das construções usuais de *loop* (KIRK, 2011, p. 34; GLASKOWSKY, 2009, p. 17). Uma execução de um *kernel* gera N diferentes execuções em paralelo por N *threads* CUDA, em oposição a apenas uma vez em uma função serial.[17] Um *thread* é como uma iteração de um *loop*. Por exemplo, tratando-se de uma manipulação de imagem, um *thread* pode operar em um *pixel*, enquanto todos os *threads* juntos (*kernel*) podem operar sobre uma imagem inteira. Uma execução de um programa CUDA é mostrada na Figura 2-9, na qual o código sequencial é executado no *host*, enquanto o código paralelo é executado no *device*.

[17] NVIDIA CUDA C Programming Guide, Version 3.2, 22/10/2010.

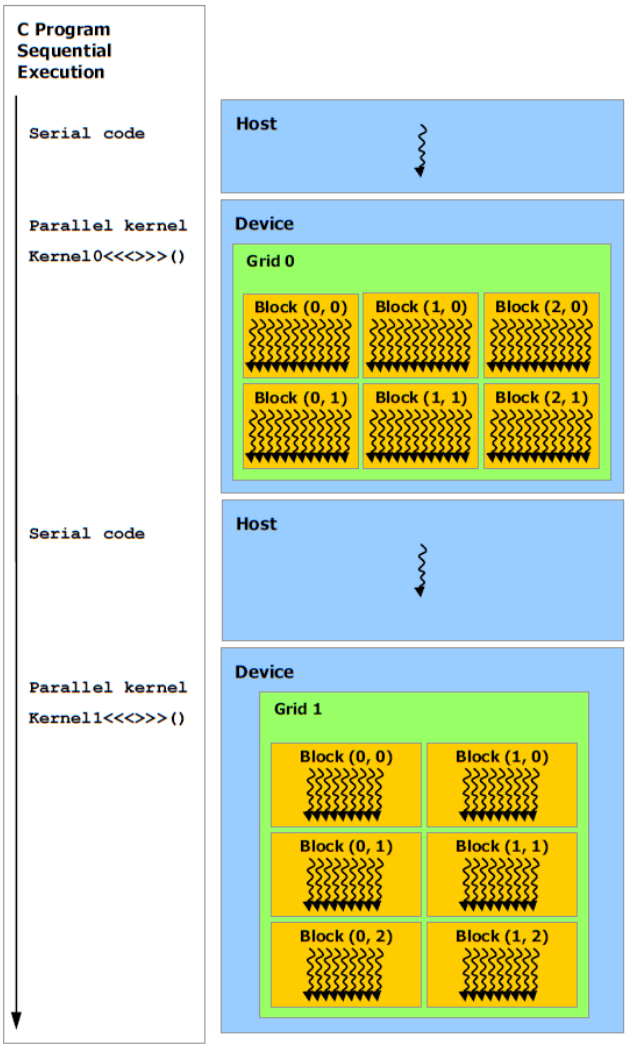

Figura 2-9 – Código sequencial, *host* e paralelo *device*
Fonte: NVIDIA CUDA C Programming Guide, Version 3.2, 22/10/2010, figura 1-4

Um *kernel* é composto por vários *threads*, até 1.536 *threads*, organizados em um *grid* de blocos de *thread*. Todos os *threads* em um mesmo bloco executam a mesma função do *kernel* (KIRK, 2011, p. 48), que será executado em um único SM; por isso, dentro do bloco, *threads* podem cooperar e compartilhar a memória. Blocos de *threads* podem coordenar o uso de memória compartilhada global entre si, mas podem executar em qualquer ordem, simultânea ou sequencialmente.

Blocos de *thread* e *threads* possuem, cada um, identificadores (*blockIdx* e *threadIdx*), que especificam sua relação com o *kernel* e identificam qual parte apropriada dos dados devem processar. Estas identificações são utilizadas dentro de cada *thread* como índices para a sua respectiva entrada e saída de dados, as localizações de memória partilhada, e assim por diante. Esta hierarquia de dois níveis, apresentada na Figura 2-10, é atribuída pelo sistema de *runtime* CUDA.

Organização de *threads*

Um *grid* é composto de blocos que são completamente independentes.

Um bloco é composto de *threads* que podem se comunicar dentro de seu próprio bloco.

Trinta e dois *threads* formam um *warp*.

Instruções são emitidas por *warp*.

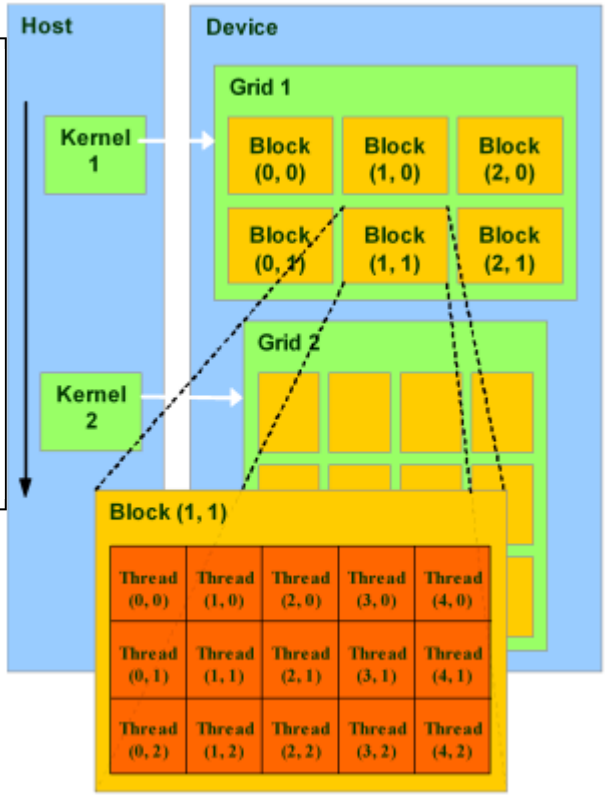

Figura 2-10 – Organização dos *threads*
Fonte: BUTLER; LUSK, 1992, figura 4

Outra significativa melhoria no desempenho da arquitetura Fermi em relação às anteriores é a capacidade de executar *kernels* simultâneos de uma mesma aplicação. Cada *kernel* será distribuído a um ou mais SMs no *device*. Este recurso evita a situação em que um *kernel* só é capaz de utilizar uma parte do dispositivo, e o resto não é usado (GLASKOWSKY, 2009, p. 14). Uma limitação é que concorrentes *kernels* devem pertencer ao mesmo programa. Fermi ainda não permite gerenciar o nível de aplicativo do paralelismo (HALFHILL, 2009, p. 15). A Figura 2-11 compara duas funções: uma escrita em código C *standard* e a outra escrita em código C paralelo CUDA.

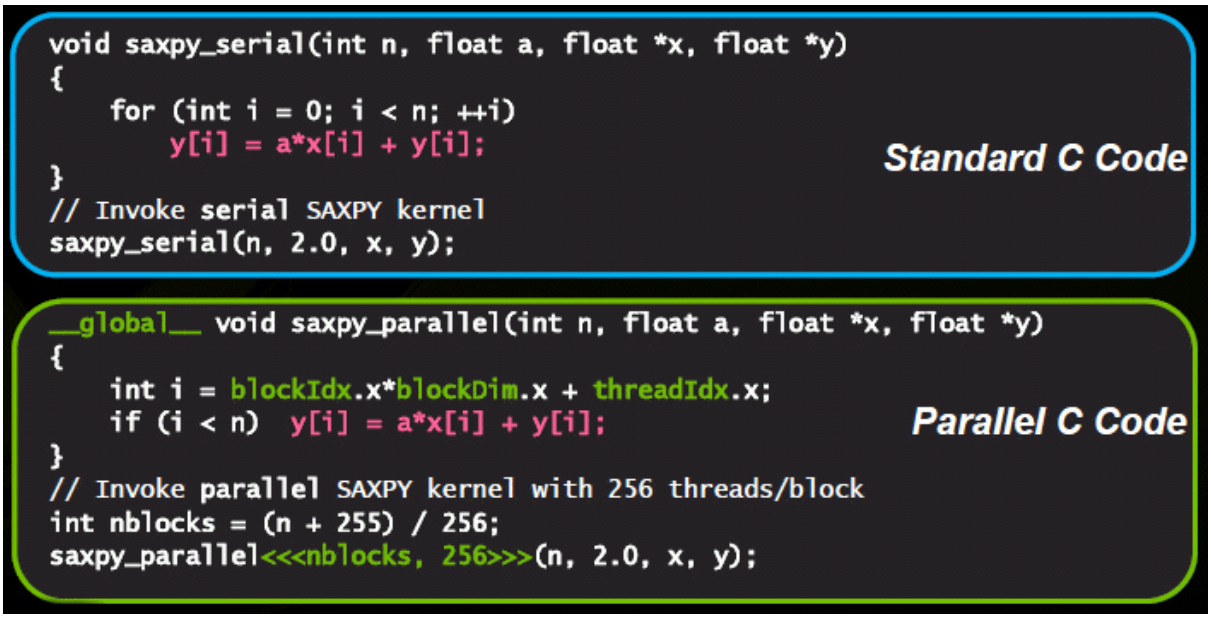

Figura 2-11 – Funções serial e paralela usando CUDA
Fonte: HALFHILL, 2009, figura 8

2.4 MPI – *Message Passing Interface*

Segundo Geist *et al*. (1994) e Sunderram (1990) (GEIST *et al.*, 1990; SUNDERRAM, 1990), por muito tempo o pacote de comunicação mais popular para multicomputadores foi o PVM (*Parallel Virtual Machine*). Contudo, nos últimos anos ele vem sendo substituído em grande parte pela MPI (*Message Passing Interface*). Comparativamente, a MPI é muito mais rica, mais complexa, tem mais chamadas de biblioteca, possui mais opções e mais parâmetros por chamadas do que a PVM.

MPI é uma especificação de biblioteca padrão para troca de mensagens. Usa o paradigma de programação paralela por troca de mensagens e pode ser usada em *clusters* ou em redes de estações de trabalho. Esta especificação é para uma biblioteca de interface. MPI é uma especificação, não uma implementação; existem várias implementações de MPI, e todas as operações MPI são expressas como funções, sub-rotinas ou métodos, de acordo com as ligações de linguagem apropriada, o que, para C, C++, Fortran-77, e Fortran-95, são parte do padrão MPI.

As principais vantagens de se estabelecer um padrão de transmissão de mensagens são a portabilidade e a facilidade de uso. Os benefícios da padronização são visivelmente evidentes quando se considera um ambiente de comunicação de

memória distribuída em que o nível de rotinas mais elevado e/ou abstrações é construído em cima de nível mais baixo de rotinas de transmissão de mensagens. Além disso, a definição de um padrão de transmissão de mensagens oferece aos fornecedores uma base claramente definida de conjunto de rotinas que possam implementar de forma eficiente, ou, em alguns casos, fornecer suporte de *hardware* para reforçar, assim, a escalabilidade.

Vários sistemas de domínio público têm demonstrado que um sistema de transmissão de mensagens pode ser implementado de forma eficiente e portável. Definir tanto a sintaxe como a semântica de um núcleo padrão de rotinas de biblioteca que será útil a um vasto leque de utilizadores e, de forma eficiente, implementável em uma ampla gama de computadores, era essencial à criação de um padrão de transmissão de mensagens. Este esforço tem sido realizado desde os anos 1990 pelo Fórum *Message Passing Interface* (MPI).

O Fórum MPI procurou fazer uso das características mais relevantes de um número existente de sistemas de transmissão de mensagens, em vez de selecionar um deles e adotá-lo como padrão. Assim, o MPI foi fortemente influenciado pelo trabalho na IBM TJ Watson Research Center (BALA; KIPNIS, 1992), NX Intel/2 (PIERCE, 1988), Express,[18] Ncube Vertex[19] e PARMACS (BOMANS; HEMPEL, 1990; CALKIN *et al.*, 1994). Outras contribuições importantes vieram de Zipcode (SKJELLUM; LEUNG, 1990; SKJELLUM *et al.*, 1992), Chimp,[20-21] PVM (BEGUELIN *et al.*, 1993; DONGARRA *et al.*, 1993), Chameleon (GROPP; SMITH, 1993) e PICL (GEIST *et al.*, 1990). A Figura 2-12 retrata diversos sistemas que contribuíram para o padrão MPI.[22]

[18] Parasoft Corporation, Pasadena, CA. Express User's Guide, version 3.2.5 edition, 1992.
[19] nCUBE Corporation. nCUBE 2. Programmers' Guide, r2.0, December 1990.
[20] Edinburgh Parallel Computing Centre, University of Edinburgh. CHIMP Concepts, June 1991.
[21] Edinburgh Parallel Computing Centre, University of Edinburgh. CHIMP Version 1.0 Interface, May 1992.
[22] Disponível em: http://www.mpi-forum.org/docs/mpi-11-html/node2.html. Acessado em: 20 out. 2011.

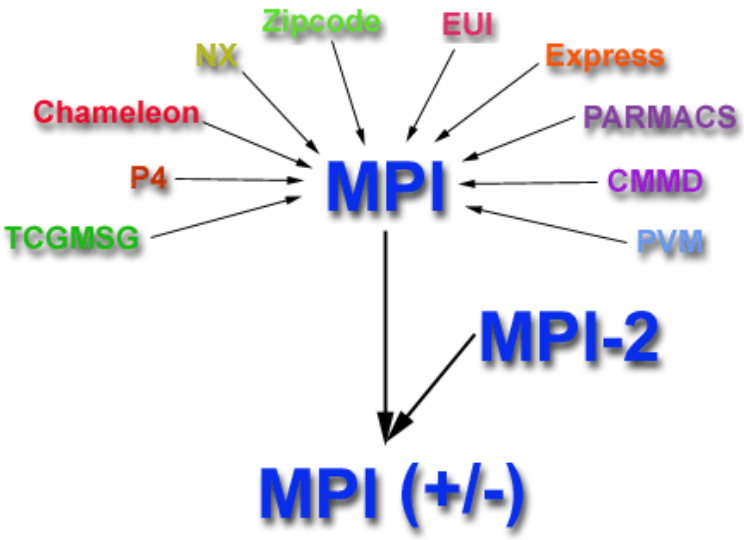

Figura 2-12 – Diversos trabalhos precederam a implementação da MPI
Fonte: https://computing.llnl.gov/tutorials/mpi/. Acessado em: 22 out. 2011.

O Fórum MPI identificou algumas deficiências críticas existentes de sistemas de transmissão de mensagens em áreas com *layouts* de dados complexos ou suporte para modularidade e comunicação segura. Isto levou à introdução de novas funcionalidades no MPI.

2.4.1 A evolução do MPI

O padrão MPI foi definido pelo Fórum MPI,[23] através de um processo aberto por uma comunidade de fornecedores de computação paralela, cientistas da computação e desenvolvedores de aplicativos. A versão original da MPI, denominada MPI-1, ficou disponível em maio de 1994 e foi atualizada pelo Fórum em setembro de 2009, para versão MPI-2.2. Eis uma breve cronologia desta evolução:

- MPI resultou dos esforços de numerosos indivíduos e grupos ao longo de um período de dois anos, entre 1992 e 1994.
- Dos anos 1980 até o início dos anos 1990: Computação paralela e memória distribuída começam a se desenvolver, assim como um número de ferramentas de *software* incompatíveis para escrever estes programas – geralmente com equilíbrio

[23] Disponível em: http://www.mpiforum.org. Acessado em: 20 dez. 2011.

entre portabilidade, funcionalidade, desempenho e preço. Surge o reconhecimento da necessidade de um padrão.

● Abril de 1992: *Workshop* sobre Normas *de Message Passing* em um ambiente de memória distribuída, patrocinado pelo Centro de Pesquisa em Computação Paralela, em Williamsburg, na Virginia. Neste *workshop*, os recursos básicos essenciais para uma interface de passagem de mensagens padrão foram discutidos, e um grupo de trabalho foi criado para continuar o processo de normalização.[24]

● Novembro de 1992: O Grupo de Trabalho se reúne em Minneapolis, a proposta do projeto MPI (MPI-1) da ORNL é apresentada, e o Grupo adota procedimentos e organização para formar o Fórum MPI. O Fórum é composto por um grupo de mais de 60 pessoas de 40 organizações, representando fornecedores de sistemas paralelos, usuários industriais, laboratórios de investigação industrial e nacional, e universidades, principalmente dos Estados Unidos e da Europa.

● Novembro de 1993: Conferência Supercomputing 93 – o projeto MPI padrão é apresentado.

● Maio de 1994: A versão final do projeto é lançada.[25]

● Em 1996: A MPI-2 foi finalizada, ampliando a capacidade da versão MPI-1 com a adição de processos dinâmicos, acesso a memória remota, comunicação coletiva sem bloqueio, suporte para E/S escalável, processamento em tempo real e muitas outras funções.

Hoje, as implementações de MPI são uma combinação de MPI-1 e MPI-2. Atualmente, o Fórum MPI trabalha no padrão MPI-3.

2.4.2 Razões para usar o MPI

O paradigma de passagem de mensagens tem atraído muitos usuários de MPI devido à sua ampla portabilidade. Programas escritos desta forma são capazes de executar em multiprocessadores de memória distribuída, redes de estações de trabalho e combinações de todos estes. Além disso, implementações que usam

[24] **MPI:** A Message-Passing Interface Standard – Version 2.2. Message Passing Interface Forum. Tennessee: University of Tennessee, September 2009.
[25] Disponível em: http://www-unix.mcs.anl.gov/mpi. Acessado em: 20 dez. 2011.

memória compartilhada, como os processadores *multi-core* e as arquiteturas híbridas, são possíveis. O paradigma não se tornará obsoleto pela combinação das arquiteturas compartilhadas e memória distribuída ou pelo aumento na velocidade da rede, sendo possível e útil para a implementação desta norma em uma grande variedade de máquinas, incluindo-se aquelas "máquinas" que consistem em coleções de outras máquinas, paralelamente ou não, ligadas por uma rede de comunicação.[26]

Decidiu-se usar a MPI neste trabalho porque foram encontradas diversas referências em pesquisas sobre este padrão, que é indicado a todos aqueles que querem escrever programas portáveis para transmissão de mensagens em C e C++. Este padrão é atraente para o público mais amplo, pois fornece uma interface simples e fácil de usar para o usuário básico, ao mesmo tempo em que não exclui a alta *performance* de transmissão de mensagens e operações disponíveis em máquinas avançadas.[27]

2.4.3 O padrão MPI

Quatro conceitos norteiam o padrão MPI: comunicadores, tipos de dados de mensagem, operações de comunicação e topologias virtuais (TANENBAUM, 2001, p. 371):

● Um comunicador é um grupo de processos mais um contexto. Um contexto é um rótulo que identifica algo, tal como uma fase de execução. Quando mensagens são enviadas e recebidas, o contexto pode ser usado para impedir que mensagens não relacionadas interfiram umas nas outras.

● Uma enorme gama de tipos é suportada, sendo também possível construírem-se outros tipos derivados desses.

● A MPI suporta um conjunto de operações de comunicação. A mais básica é usada para enviar mensagens como segue:

MPI_Send (*buffer, count, data_type, destination, tag,* comunicador)

---

[26] MPI: A Message-Passing Interface Standard. Version 2.2. Message Passing Interface Forum, September 4, 2009.
[27] *Idem*, p. 4.

Esta operação de comunicação envia ao destinatário um *buffer* com um número *count* de itens e o tipo de dados especificado. O campo *tag* rotula a mensagem de modo que o receptor possa dizer que só quer receber uma mensagem com aquele rótulo. O último campo informa em qual grupo de processo está o destinatário (o campo *destination* é apenas um índice para a lista de processos do grupo especificado). A chamada correspondente para receber a mensagem é:

MPI_Recv (*buffer, count, data_type, destination, tag*, comunicador)

Ela anuncia que o receptor está procurando uma mensagem de um certo tipo vinda de uma determinada fonte com um rótulo específico.

A MPI suporta quatro modelos básicos de comunicação: o modo síncrono, no qual o remetente não pode começar a enviar até que o receptor tenha chamado MPI_Rec; o modo *buffer*, onde não existe a restrição anterior; o modo padrão, que é implementado independente e pode ser síncrono ou *buffer*; e o modo pronto, no qual o remetente declara que o receptor está disponível (como no modo síncrono), mas não faz nenhuma verificação. Cada uma destas primitivas vem em uma versão com bloqueio ou sem bloqueio, que resulta em oito primitivas no total. A recepção tem duas variantes, com bloqueio e sem bloqueio.

A MPI suporta comunicação coletiva, incluindo *broadcast*, espalha/reúne, permuta total, agregação e barreira.

● O último conceito básico em MPI é a topologia virtual, na qual os processos podem ser organizados em topologia de árvore, anel, grade ou outra. Esta organização proporciona um meio de nomear caminhos e facilitar a comunicação.

Embora muitas características tenham sido consideradas e não incluídas neste padrão, como, por exemplo, facilidade de debugar, isto aconteceu por uma série de razões, uma das quais é a restrição de tempo que foi autoimposta para conclusão do padrão. Recursos que não são incluídos sempre podem ser oferecidos como extensões de implementações específicas. Talvez as futuras versões do MPI abordem algumas destas questões.

2.4.4 Estrutura do código com MPI

Um código MPI deve sempre ser inicializado com MPI_Init (& argc , & argv) e concluído com MPI_Finalize (). Na Figura 2-13, apresenta-se um código C, usando MPI. Observa-se que a variável *rank* controla o que as máquinas *slave* e *master* executarão. A comunicação é dita bloqueante e todas as *slaves* enviarão *sendmsg* para a máquina *master*.

```
# include <mpi.h>
# include <stdio.h>
# define ROOT 0
# define MSGLEN 100
int main (int argc , char ** argv )
{
   int i, rank , size , nlen , tag = 999;
   char name [ MPI_MAX_PROCESSOR_NAME ];
   char recvmsg [ MSGLEN ], sendmsg [ MSGLEN ];
   MPI_Status stats ;
   MPI_Init (& argc , & argv );
   MPI_Comm_rank ( MPI_COMM_WORLD , & rank );
   MPI_Comm_size ( MPI_COMM_WORLD , & size );
   MPI_Get_processor_name (name , & nlen );
   sprintf ( sendmsg , " Hello world ! I am process %d out of %d on
%s\n", rank , size , name );
   if ( rank == ROOT ) {
      printf (" Message from process %d: %s", ROOT , sendmsg );
      for (i = 1; i < size ; ++i) {
          MPI_Recv (& recvmsg , MSGLEN , MPI_CHAR , i, tag ,
MPI_COMM_WORLD , & stats );
          printf (" Message from process %d: %s", i, recvmsg );
      }
   } else {
          MPI_Send (& sendmsg , MSGLEN , MPI_CHAR , ROOT , tag ,
MPI_COMM_WORLD );
   }
   MPI_Finalize ();
   return 0;
}
```

Figura 2-13 – Exemplo de programa usando MPI com transmissão de mensagens síncrona

2.5 Proposta do trabalho – *Cluster* GPU/MPI

Em computação, *cluster* é o uso de vários computadores, normalmente PCs ou estações de trabalho Linux/Unix, que podem contar com vários dispositivos de armazenamento e interconexões redundantes para formar o que aparece para os usuários como um único sistema, com grande capacidade de processamento e alta disponibilidade. Usuários de *clusters* sugerem que a abordagem pode ajudar uma empresa a alcançar 99,99% de disponibilidade, em alguns casos. Uma das ideias principais da computação *cluster* consiste no fato de que, para o mundo exterior, o *cluster* parece ser um único sistema.

Computação em *cluster* pode ter foco na alta disponibilidade ou então no processamento paralelo. No processamento paralelo, objeto deste trabalho, a tarefa a ser realizada é dividida entre suas diversas máquinas, visando à sua realização em um tempo mais curto.

Neste trabalho, utiliza-se um *cluster* formado por seis nós (denominação dada a cada microcomputador do *cluster*) interligados por rede, utilizando protocolo MPI para comunicação entre os processadores. O Sistema Operacional adotado foi a versão Linux Fedora 16. Cada nó possui processadores i7-960[28], 8M Cache, 3.2GHz, 4GB de memória RAM, 1 TB de *hard disk* e duas GPUs GTX480. Apenas quatro nós foram utilizados no presente trabalho, totalizando oito GPUs. Nosso objetivo foi desenvolver programas em linguagem C, visando investigar os ganhos e as eventuais limitações da utilização da computação paralela em um *cluster* de computadores, usando processamento em suas GPUs. Para tal, foram desenvolvidas diversas versões de programas para realizar as mesmas tarefas: uma versão sequencial e outras paralelas, utilizando de uma a oito GPUs do *cluster*. O *cluster* utilizado neste trabalho é apresentado na Figura 2-14.

[28] Sobre i7, confira-se o *site* http://ark.intel.com/products/37151/Intel-Core-i7-960-Processor-(8M-Cache-3_20-GHz-4_80-GTs-Intel-QPI)

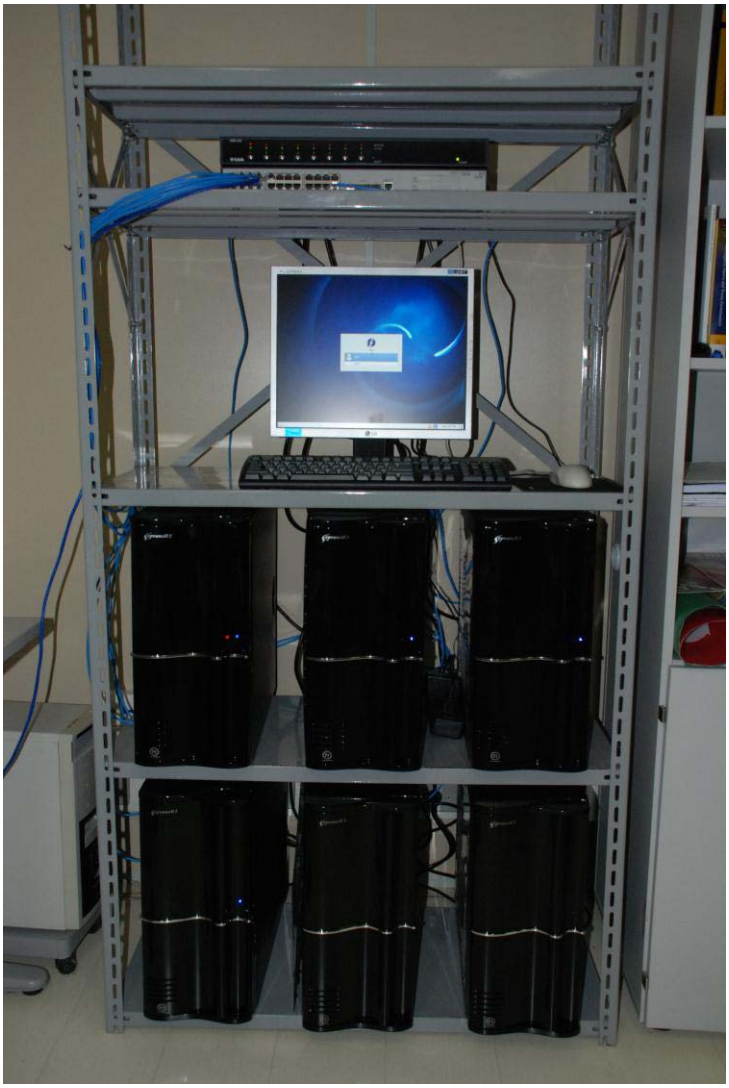

Figura 2-14 – *Cluster* do LIAA – IEN – RJ

Um diagrama esquemático da arquitetura do *cluster* de GPU é exibido logo abaixo, na Figura 2-15:

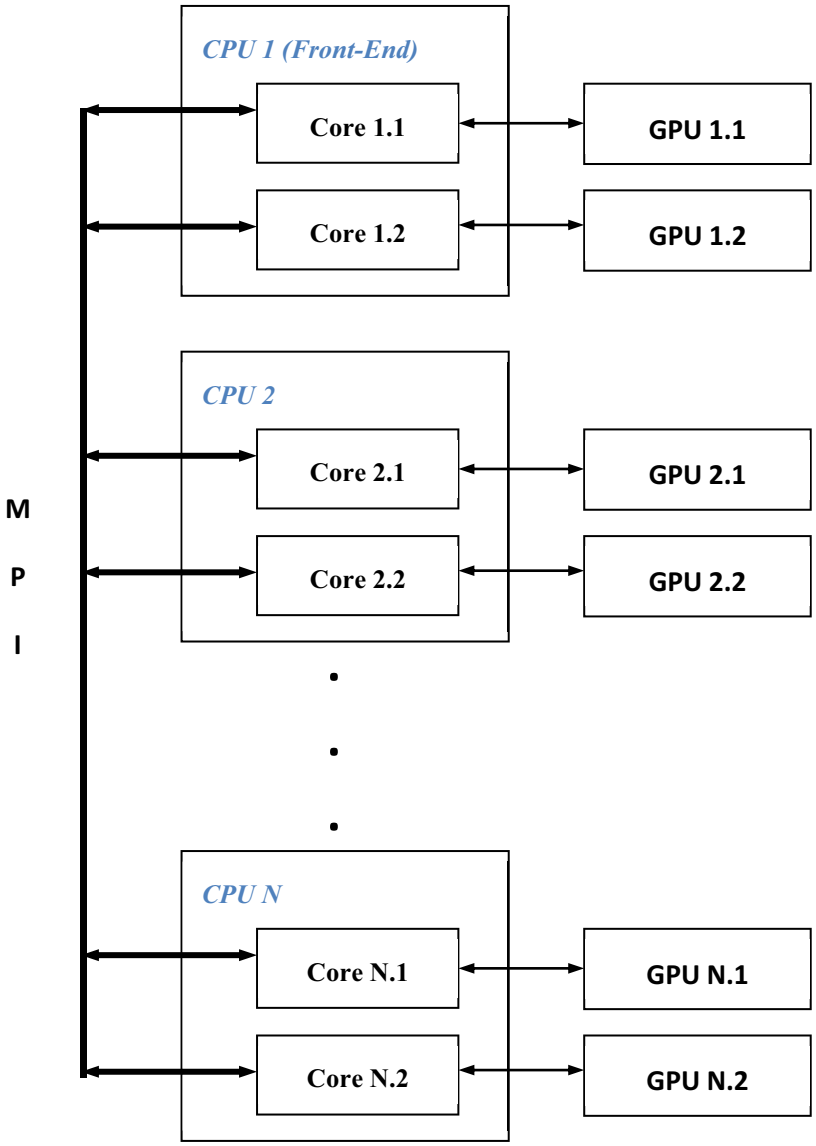

Figura 2-15 – Diagrama esquemático do *cluster* de GPU utilizado

Cada GPU é associada a um núcleo (processador), tornando seu controle (utilização) independente. Desta forma, através da utilização "*multithreading*" (programas com controles de tarefas independentes), obtém-se o paralelismo entre as GPUs de um mesmo nó. Através da utilização do MPI, N nós podem realizar tarefas em paralelo.

Em um dos nós (*front-end*), também utilizado para processamento, a tarefa é dividida e submetida a cada nó que, após execução, retorna seu resultado para o *front-end*, que "colapsa" as informações recebidas, gerando o resultado final para a simulação.

3 SIMULAÇÃO DO TRANSPORTE DE NÊUTRONS ATRAVÉS DE UM *SLAB* UTILIZANDO O MÉTODO MONTE CARLO

3.1 O transporte de nêutrons

Uma vez que os nêutrons possuem carga elétrica nula, eles não são afetados pelas forças coulombianas dos elétrons ou pela carga positiva do núcleo e são capazes de atravessar nuvens de elétrons e interagir diretamente com os núcleos, ao contrário do que ocorre com outras radiações, por exemplo, os raios gama. Os nêutrons podem interagir com os núcleos das seguintes maneiras:

- Espalhamento elástico
- Espalhamento inelástico
- Captura radiativa
- Reações com produção de partículas carregadas
- Reações com produção de nêutrons
- Fissão

Delimitando um escopo para o problema, fizeram-se algumas considerações: o fluxo de nêutrons é monoenergético e uniforme, e atravessa uma placa anisotrópica (*slab*) de espessura *x*; além disso, o fluxo de nêutrons atinge normalmente a superfície da placa, como se vê na Figura 3-1.

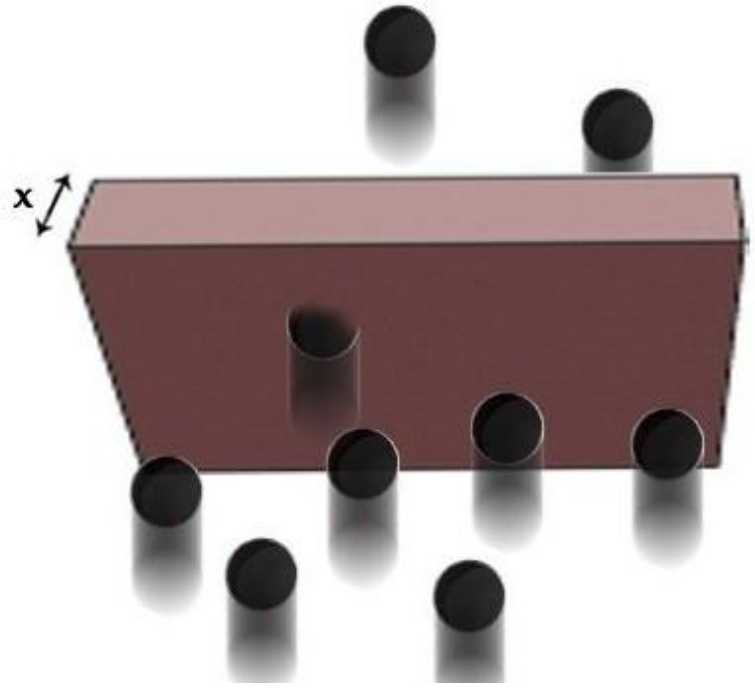

Figura 3-1 – *Slab* diante de um feixe de nêutrons
Fonte: MORAES, 2012.

O resultado da interação de um grande número de partículas deve ser levado em conta em muitos problemas físicos. As leis de interações elementares (leis microscópicas) são conhecidas a partir de experimentos ou podem ser previstas teoricamente. Entretanto, as características macroscópicas da matéria (por exemplo, a densidade) são conhecidas somente através de métodos empíricos (SHREIDER, 1964, p. 84).

O método clássico de solução de tais problemas baseia-se em equações que são satisfeitas pelas características macroscópicas. Equações de difusão são um exemplo deste tipo de abordagem, e métodos numéricos estão disponíveis para sua solução. Em outros problemas, é necessário usar as equações de transporte ou a equação cinética, para os quais métodos numéricos de solução têm sido desenvolvidos apenas em casos simples. Para muitos problemas, as equações macroscópicas não estão disponíveis (SHREIDER, 1964, p. 84).

Entretanto, o Método Monte Carlo pode ser usado para o cálculo aproximado sem recorrer a equações macroscópicas, sendo capaz de produzir uma quantidade considerável de informações como, por exemplo, o comportamento assintótico, relações aproximadas, e assim por diante. Monte Carlo é um método numérico e

deve ser comparado com outros métodos numéricos para a solução da equação macroscópica, e não com o método de equações macroscópicas em si. Em muitos problemas e, particularmente, em problemas complicados, o Método Monte Carlo tem muitas vantagens em comparação com o método clássico numérico, além do fato de que em muitos casos as equações macroscópicas não são conhecidas (SHREIDER, 1964, p. 84).

3.2 Método Monte Carlo

O nome e o desenvolvimento sistemático do Método Monte Carlo datam de cerca de 1944 (SKJELLUM; LEUNG, 1990). No entanto, existe uma série de casos isolados já na segunda metade do século XIX. O uso do Método Monte Carlo como ferramenta de investigação é o resultado do trabalho sobre a bomba atômica durante a Segunda Guerra Mundial, num projeto denominado *Manhatan*. O nome "Monte Carlo" foi dado por Metropolis, inspirado no interesse do pesquisador Ulam por pôquer durante o projeto *Manhatan*, devido à similaridade da simulação estatística de jogos de azar e ao fato de a capital de Mônaco ser conhecida como a capital mundial dos jogos de azar. No entanto, o desenvolvimento sistemático teve de aguardar o trabalho de Harris e Herman Kahn, em 1948.

O Método Monte Carlo é um método estocástico que, por sua vez, utiliza uma sequência de números aleatórios (YORIYAZ, 2010) para realizar a simulação, baseando-se nas leis de probabilidade e estatística para caracterizar um processo físico. Simulações estatísticas contrastam com métodos convencionais de discretização das variáveis do processo físico em estudo, que tipicamente são aplicados em sistemas de equações diferenciais parciais ou ordinárias.

3.3 Aplicação do Método Monte Carlo ao problema

Suponha-se que um feixe de partículas entre em uma região $\boldsymbol{\sigma}$ de um meio. Considere-se uma destas partículas e suponha que a distribuição de caminhos livres é conhecida, de modo que é possível escolher um livre caminho médio e encontrar o ponto de colisão desta partícula com um átomo nesta região (SHREIDER, 1964).

Se a região contém uma mistura de meios, então, é possível determinar com qual tipo particular de átomo a partícula interage, porque as probabilidades de colisão são proporcionais aos montantes dos vários átomos presentes (SHREIDER, 1964).

A partícula incidente pode deixar de existir, isto é, ela pode ser absorvida ou ser espalhada, ou seja, pode adquirir direção e energia diferentes. As probabilidades de várias interações entre partículas incidentes de um tipo particular e os átomos-alvo são conhecidas: elas são caracterizadas pelas seções de choque macroscópicas ou microscópicas.

A seção de choque macroscópica total (Σ_t) é a probabilidade, por unidade de comprimento, de um nêutron sofrer espalhamento ou captura (TAUHATA, 2003, p. 85). A probabilidade de interação de nêutrons com o núcleo de um átomo é representada pela chamada seção de choque microscópica (**σ**), que está relacionada à área projetada do núcleo de um átomo.[29]

É de nosso interesse conhecer a história de cada nêutron. Na física de nêutrons, a interação destes com núcleos é descrita pelas seções de choque microscópicas. Suponha-se que um feixe homogêneo de nêutrons incide perpendicularmente sobre uma camada monatômica de unidade de área, e *n* é o número de átomos na camada por unidade de área. Se as frações de nêutrons que participam de uma interação é ***d***, então, a seção de choque microscópica **σ** do núcleo para uma interação particular é dada por (SHREIDER, 1964, p. 92).

$$\sigma = \frac{d}{n} \tag{3.1}$$

As seções de choque microscópicas mais frequentemente encontradas são as seguintes:

σ_s *seção de choque microscópica de espalhamento* $\sigma_s = \sigma_{se} + \sigma_s$,

σ_c *seção de choque microscópica de captura*

σ_f *seção de choque microscópica de fissão*

σ_a *seção de choque microscópica de absorção* $\sigma_a = \sigma_f + \sigma_c$

σ_t *seção de choque microscópica total* $\sigma_t = \sigma_s + \sigma_c + \sigma_f$

[29] Princípios Básicos de Segurança e Proteção Radiológica. Universidade Federal do Rio Grande do Sul, p. 85, set. 2006.

As seções de choque macroscópicas são definidas como o produto

$$\Sigma = \rho\sigma \tag{3.2}$$

onde ρ é a densidade nuclear, isto é, o número de núcleos por unidade de volume.

No caso de um meio não homogêneo:

$$\Sigma \;\; = \; \Sigma_{(1)} + \; \Sigma_{(2)} + \cdots + \; \Sigma_{(m)} \tag{3.3}$$

A razão entre as seções de choque macroscópicas com a seção de choque total caracteriza as probabilidades de as várias interações ocorrerem quando um nêutron colide com o núcleo. Elas são usadas em determinação da história do nêutron. Simplificadamente, considera-se um feixe uniforme de elétrons, que se assumirá ser homogêneo, e que incidirá sobre o *slab* (placa plana paralela que não contém material físsil), de modo que a seção de choque macroscópica total é dada pela Equação

$$\Sigma_{\mathrm{T}} = \; \Sigma_s + \; \Sigma_c \, , \tag{3.4}$$

Então a probabilidade de o nêutron espalhar é: $\frac{\Sigma_s}{\Sigma_{\mathrm{T}}}$, (3.5)

A probabilidade de o nêutron ser capturado é: $\frac{\Sigma_c}{\Sigma_{\mathrm{T}}}$ (3.6)

Antes de se aplicar o Método Monte Carlo, atribui-se numa escala arbitrária, representada na Figura 3-2, um comprimento de três unidades para a probabilidade de espalhamento Σ_s. Para determinar o tipo de interação, encontra-se um valor de um número aleatório Y e se determina em qual dos dois intervalos ele se encontra.

Se $\mathrm{Y} < \frac{\Sigma_s}{\Sigma_t}$

Isto significa que ocorreu um espalhamento. Caso contrário, ocorreu uma absorção ($\sigma_a = \sigma_c$) do nêutron.

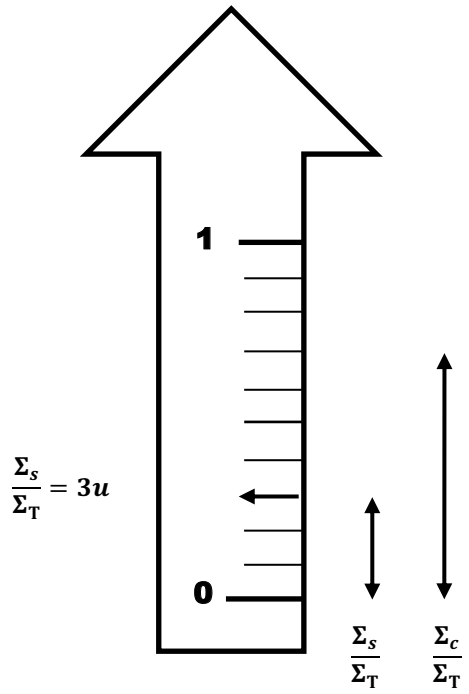

Figura 3-2 – Escala arbitrária para efeito de comparação da seção macroscópica
Fonte: MORAES, 2012.

O livre caminho médio de um nêutron é uma quantidade aleatória. A lei da distribuição de caminhos livres é obtida através da fórmula

$$\mathrm{P}[l < x] = 1 - e^{-\int_0^x \Sigma_\mathrm{T}\, ds} \tag{3.7}$$

onde s é a distância a partir da colisão precedente ao longo da direção de movimento do nêutron. A densidade de distribuição é dada por

$$\mathrm{P}(x) = \Sigma_\mathrm{T} e^{-\int_0^x \Sigma_\mathrm{T}\, ds} \tag{3.8}$$

Em um meio homogêneo no qual Σ_T é independente de s, o livre caminho médio é dado por

$$\overline{\lambda} = \frac{1}{\Sigma_\mathrm{T}} \tag{3.9}$$

Em um meio homogêneo, o livre caminho pode ser determinado pelo Método Monte Carlo com o auxílio da equação

$$\lambda = -\frac{1}{\Sigma_T} \ln Y \tag{3.10}$$

Onde Y é um número aleatório. Se o livre caminho médio λ é tomado como a unidade de comprimento, então

$$\lambda = -\ln Y \tag{3.11}$$

O livre caminho médio de um nêutron pode ser definido para qualquer tipo de interação, por exemplo, absorção ou espalhamento. Esta regra é válida para todas as fórmulas anteriores, exceto para Σ_T, que deve ser substituída pela seção de choque de interação apropriada (Σ_a, no caso de absorção ou Σ_s no caso de espalhamento).

4 IMPLEMENTAÇÃO DAS SOLUÇÕES PARA O PROBLEMA

Desenvolveram-se diversas implementações das soluções com o objetivo de se avaliar o ganho de *performance* de cada implementação. Partindo-se de uma solução sequencial escrita em linguagem C para o Método Monte Carlo para simulação de transporte de nêutrons, desenvolveram-se versões paralelas para uma GPU, duas GPUs e uma solução para múltiplas GPUs.

4.1 Solução sequencial

Apresenta-se um algoritmo da solução sequencial na Figura 4-1. Em todas as soluções fizeram-se necessárias: a definição do número de histórias de nêutrons (NH) a se tratar, a espessura do *slab* e a definição das características físicas (Σ_s e Σ_a) do meio considerado em cada experimento (Alumínio, Cádmio ou Água).

```
1.      Inicializa NH ← 1E10
2.      Inicializa dados físicos do Meio

Função HistoryNeutronCPU ( Meio ) {
    :
    // Simulação Monte Carlo
}
Int main(void) {

    3.  Inicializa count _cpu ← 0
    4.  Count_cpu ← HistoryNeutronCPU( Meio )
    5.  Fator de transmissão ← Count_cpu/NH
    6.  Exibe de Fator de transmissão

}
```

Figura 4-1 – Algoritmo da solução sequencial

As Tabelas 4-1 e 4-2 mostram as características físicas adotadas em todas as soluções.

Tabela 4-1 – Seções de choque macroscópicas

Meio	Alumínio	Cádmio	Água
Σ_a (cm^{-1})	0,015	114	0,022
Σ_s (cm^{-1})	0,084	0,325	3,450

Fonte: DUDERSTADT; HAMILTON, 1975, p. 606.

Tabela 4-2 – Valores das características físicas adotadas[30]

Meio	Alumínio	Cádmio	Água
x (cm)	10,0	0,01	10,0
Fator de transmissão (teórico)	0,8607	0,3198	0,8025

A Figura 4-2 ilustra um trecho do código da solução sequencial incluindo a função HistoryNeutronGPU e o programa principal. Observa-se, nesta solução, o número de história de nêutrons (NH) definido na linha 5. Objetivamente, ao final da execução da função HistoryNeutronCpu (linha 8), simulou-se quantos nêutrons conseguiram atravessar o *slab* e, assim, encontrou-se o **fator de transmissão**. Nesta solução todo o processamento é realizado exclusivamente por uma única CPU em um dos PCs do *cluster*, ao contrário das soluções paralelas onde o número de história de nêutrons foi dividido pelo número de GPUs do *cluster*.

[30] Deseja-se encontrar o fator de transmissão (fração de nêutrons que atinge a espessura x), cuja solução teórica, segundo as hipóteses já colocadas é dada por: $I(x)/I_o = \exp(-\Sigma_a.x)$.

```
1  /***********************************
2  SOLUÇÃO SEQUENCIAL
3  ***********************************/
4  // Define Nro de Histórias
5  #define NH 1E10
6
7  //Função sequencial para o Cálculo História Nêutron CPU
8  unsigned long long  HistoryNeutronCpu ( TpMeio Meio )
9  {
10   // Cálculo do livre caminho médio e probabilidade de absorção
11   MeanFreePath= (1 / (Meio.Sigma_S + Meio.Sigma_A ) );
12   PA        = ( Meio.Sigma_A/(Meio.Sigma_S+Meio.Sigma_A ) );
13
14  // Inicialização  série  randômica
15   srand( time(NULL) );
16
17  // Loop com NH histórias
18   for (n=0; n<NH; n++) {
19         x = (float)0;
20     do {
21       // Cálculo caminho do nêutron
22       x += -MeanFreePath*log( Rand() );
23        if (x > Meio.Width) {
24          tot++;
25           break;
26       }
27        // Testa se foi absorvido
28       if ( Rand() < PA ) break;
29     } while( true );
30   } //for
31   return ( tot );
32 }
33
34 int main ()
35 {
36   :
37   fator = HistoryNeutronCpu ( Meio ) / NH;
38   printf( "CPU Fator calculado em %f\n", ( double ) fator );
39 }
```

Figura 4-2 – Codificação da solução sequencial

4.2. Solução paralela 1 GPU

Partindo-se da solução sequencial, construiu-se uma solução paralela, usando CUDA que executasse em uma única GPU. Um programa escrito em CUDA é composto por uma parte sequencial, uma parte paralela e, novamente, uma parte sequencial. Quando a parte paralela conhecida como *kernel* é executada, o maior número de *threads* é gerado e a execução passa para o *device* (GPU). O conjunto de *threads* gerados por um *kernel* durante uma chamada é conhecido como *grid* (KIRK, 2011, p. 35). A Figura 4-3 ilustra a execução de dois *grids* de *threads.*

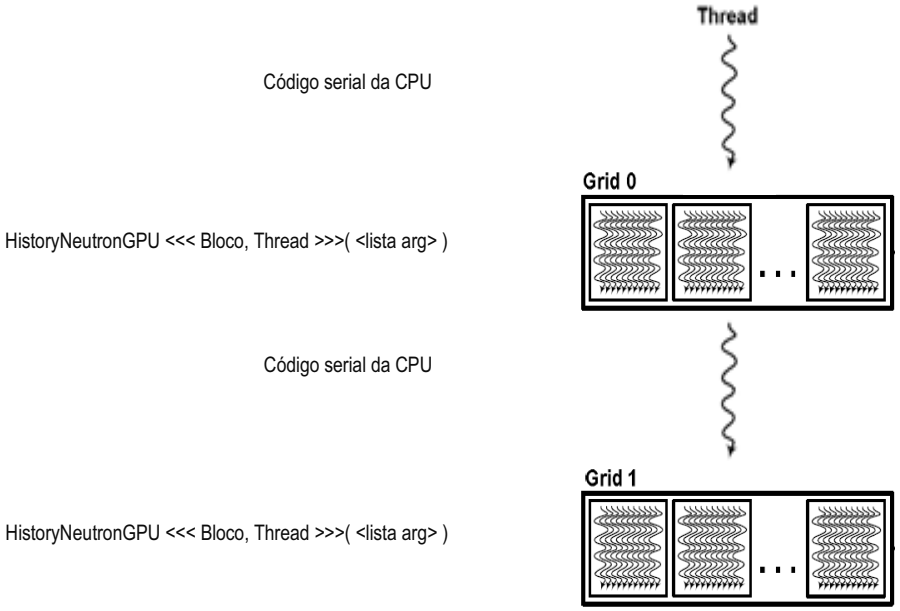

Figura 4-3 – Sequência de execução CUDA

Uma função *kernel* especifica o código a ser executado por todos os *threads* de um mesmo *grid*. Sendo assim, faz-se necessário o uso de coordenadas únicas para distinção dos *threads* e em que parte da memória elas devem acessar a estrutura de dados sobre as quais devem atuar. Criadas pelo CUDA Runtime System e acessíveis apenas pela função *kernel*, as variáveis pré-definidas threadIdx.x, threadIdx.y e threadIdx.z são atribuídas aos *threads* que pertencem a um bloco de coordenadas blockIdx.x e blockIdx.y.

Com uma lógica muito semelhante à do sequencial, nesse código observam-se algumas particularidades da plataforma CUDA. Na figura 4-4, vê-se parte do algoritmo da função *kernel* HistoryNeutronsGPU, e na linha 8, a identificação do *thread.*

```
1  //inicializa NE (Nro total de Threads)
2  NE = NroBlocos * NroThreads
3
4  //kernel para HistoryNeutronGPU
5  __global__void HistoryneutronGPU (seed, Meio, count_GPU)
6  {
7    // ID do Thread iD
8    idx = blockIdx.x * blockDim.x + threadIdx.x;
9    //Inicializa curand(seed+idx)
10     :
11   //Monte Carlo
12   for ( i < (NH/NE+1); i++ ) {
13         :
14        //Calcula total
15         :
16   }
17   //Sincroniza Threads
18   Count_GPU[idx] ← total
19 }
```

Figura 4-4 – Identificação dos *threads* no algoritmo do *kernel* HistoryneutronGPU

Enquanto na solução sequencial o *loop for* itera NH vezes, na solução paralela ele itera NH/NE+1 vezes todo o processo, ou seja, ter-se-á NE vezes menos iterações no laço em razão de os *threads* executarem o *kernel* em paralelo, tornando o código muito mais eficiente.

A eficácia do Método Monte Carlo depende da qualidade dos números aleatórios gerados pelos algoritmos que os fornecem. A CUDA disponibiliza uma biblioteca chamada CURAND, que possibilita, eficientemente, uma geração com alta qualidade de números pseudorandom e quasirandom. Segundo *CURAND Guide* (2011), uma sequência de números pseudorandom e quasirandom é concebida por um algoritmo determinista que satisfaz a maioria das propriedades estatísticas de uma sequência verdadeiramente aleatória. A Figura 4-5 ilustra um trecho do código da solução paralela 1 GPU.

```
1  /**********************************
2  FUNÇÃO KERNEL HistoryNeutronGPU
3  **********************************/
4
5  __global__ void HistoryNeutronGPU ( const unsigned long long seed, TpMeio m, unsigned long long *count )
6  {
7     float  x, rnd;
8     float  MeanFreePath, PA;
9     unsigned long long tot = 0;
10    unsigned long long idx = blockIdx.x * blockDim.x + threadIdx.x;
11
12    // COPIA state PARA MEMÓRIA LOCAL    curandState localState;
13    curand_init( (seed + idx), idx, 0, &localState);  //  INICIALIZA  CURAND
14
15    for ( unsigned long long i = 0; i < (unsigned long long) ( NH / NE ) + 1; i++) {
16        x = (float)0;
17        MeanFreePath= (1 / (m.Sigma_S + m.Sigma_A ));
18        PA = (m.Sigma_A / (m.Sigma_S + m.Sigma_A ));
19        do {
20             rnd = curand_uniform( &localState );
21             x += -MeanFreePath * log(rnd);
22             if (x > m.Width) {
23                      tot++;
24                      break;          // Fim da História
25                      }
26            rnd = curand_uniform( &localState );
27            if ( rnd < PA ) break;
28        }
29        while( true );
30    } // Fim do for
31    count[idx] = tot;
32    __syncthreads();
33  } // Fim HistoryNeutronGPU
```

Figura 4-5 – Simulação Monte Carlo em 1 GPU

Na estrutura de um programa CUDA, *device* e *host* não dividem espaços de memória. *Device* e *host* possuem distintas memórias. Logo, para executar um *kernel* em um *device*, o desenvolvedor deve alocar memória no *device* e copiar os dados pertinentes da memória do *host* para a memória do *device*. Semelhantemente, devem-se copiar os dados correspondentes aos resultados encontrados no *device* para a memória do *host.* Nesta simulação, a etapa de transferência *host-device* não foi necessária, haja vista os dados referentes à simulação da história de cada nêutron ter sido gerada randomicamente no próprio *kernel*. Entretanto, o caminho

inverso, *device-host*, foi imprescindível. CUDA nos fornece, pelas funções da API, a função cudaMalloc() e cudaMemcpy() que, executadas no *host*, alocam e transferem dados. As etapas correspondentes à alocação e à transferência de dados estão respectivamente nas linhas 15 e 21 da Figura 4-6, onde se vê um trecho de código que completa a solução paralela 1 GPU.

```
1.  /***********************************
2.    SOLUÇÃO PARALELA 1 GPU
3.  ***********************************/
4.  int main()
5.  {
6.  TpMeio          Meio;
7.  unsigned long long  *count_h, *count_d;
8.  unsigned long long  n, count_gpu;

9.  // Inicializacao contador de historias
10. count_gpu = ( unsigned long long ) 0;

11. // Aloca array count_h no host
12. count_h = ( unsigned long long *) malloc(sizeof( unsigned long long ) * NE);
13. memset(count_h, 0, sizeof(unsigned long long) * NE);

14. // Aloca array count_d no device
15. cudaMalloc((void **) &count_d,   sizeof( unsigned long long ) * NE);

16. /* Inicializa características do meio
17. :
    :
18. // Executa o kernel
19. historyneutron_gpu<<< N_BLOCKS, N_THREADS >>>( rand(), Meio, count_d );

20. // Copia dados do device para o host
21. cudaMemcpy(count_h, count_d, sizeof( unsigned long long ) * NE, cudaMemcpyDeviceToHost);

22. /* Totaliza nro de histórias */
23. for (n = 0;n < NE; n++) count_gpu += count_h[n];

24. printf ("\n * Slab          ==> %s",Meio.slab);
25. printf ("\n Fator teórico      = %f\n", Meio.VlrTeorico);
26. printf( "\n CUDA Fator estimado = %f [error of %f]",  ( float ) count_gpu/NH, ( float ) count_gpu/NH -
    Meio.VlrTeorico );

27. return 0;
```

Figura 4-6 – Trecho do código do programa principal da solução paralela

4.3 Solução paralela 2 GPUs

Todos os nós do *cluster* utilizam processadores Intel i7-960[31] (Desktop Processor Series) com quatro cores e tecnologia HT (*hyperthreading*). O sistema operacional Fedora 16, usado em cada nó do *cluster*, é um moderno sistema SMP (multiprocessamento simétrico), que admite trabalhar com mais de um processador instalado, dividindo as tarefas entre os mesmos.

CUDA suporta uma estrutura separada do *host* (CPU do sistema) e *device* (GPU), onde o *host* executa o programa principal e o *device* atua como coprocessador. A Figura 4-7, a seguir, representa esquematicamente como se buscou aproveitar o máximo dos recursos do *hardware* disponível, dividindo-se tarefas para cada GPU de forma que pudessem ser executadas paralelamente, ou seja, usando *multithreading*.

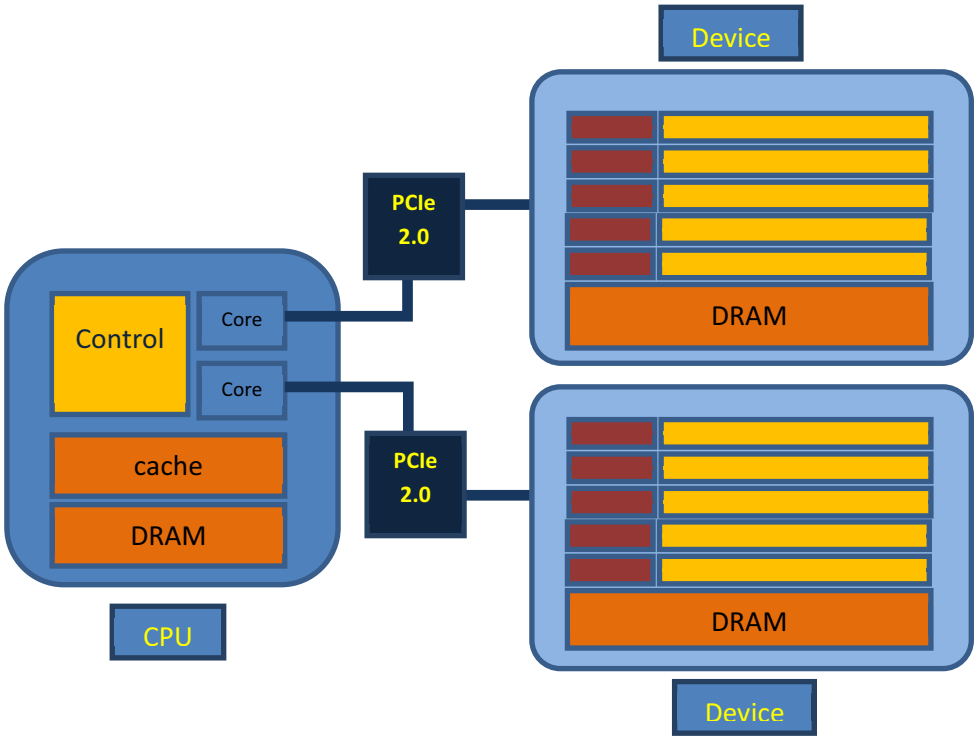

Figura 4-7 – Arquitetura *multithreading* entre CPU e 2 GPUs
Fonte: MORAES, 2012.

NVIDIACUDA Runtime API disponibiliza ao desenvolvedor do programa a possibilidade de selecionar qual *device* executará o *kernel*. Por padrão, o *device* 0 é

[31] Sobre i7, confira-se o *site* http://ark.intel.com/products/37151/Intel-Core-i7-960-Processor-(8M-Cache-3_20-GHz-4_80-GTs-Intel-QPI)

o usado, e os demais *devices* são enumerados progressivamente. Com isto, através da CUDA, pode-se escrever um código em linguagem C, onde se distribuíram tarefas, através de *threads*, para cada GPU. O trecho do código destacado na Figura 4-8 nos mostra o disparo dos *threads*.

```
/**********************
TRECHO CÓDIGO 2 GPU
**********************/
int main()
{
    //Solver config
    TGPUplan     plan[2];
    const unsigned long long DATA_N = NH;
    int GPU_N;
    unsigned long long int i, gpuBase;
    unsigned long long *h_Data;
    unsigned long long sumGPU;
    unsigned long long h_SumGPU[2];

    //OS thread ID
    CUTThread threadID[2];
    // Inicialia GPU_N
    cutilSafeCall(cudaGetDeviceCount(&GPU_N));

    // Randomiza
    srand( time(NULL) );

    // Divide NH pelo nro GPUs
    for ( i = 0; i < GPU_N; i++ )
        plan[i].dataN = DATA_N / GPU_N;

    //Ajusta dataN, caso DATA_N(NH) não seja múltiplo de GPU_N
    for ( i = 0; i < DATA_N % GPU_N; i++ )
        plan[i].dataN++;

   //Atribui intervalos de dados para GPUs323      gpuBase = 0;
   for ( i = 0; i < GPU_N; i++ ){
      plan[i].device = i;
      plan[i].h_Data = h_Data + gpuBase;
      plan[i].h_Sum = h_SumGPU + i;
      gpuBase += plan[i].dataN;
    }
    // Dispara as THREADS
    for ( i = 0; i < GPU_N; i++ )
    {
        threadID[i] = cutStartThread((CUT_THREADROUTINE)DisparaThreads, (void *)(plan + i));
    }
    cutWaitForThreads(threadID, GPU_N
```

Figura 4-8 – Pseudocódigo parte programa principal da solução 2 GPUs

Observa-se que, nessa solução, ao contrário da solução serial, o número de histórias (NH) foi dividido pelo número de GPUs (linha 25) e armazenado em um campo de uma estrutura de dados que será usada para comunicação entre as funções *host*, *threads* e *kernel* que será executado por cada *device*. Vê-se, na linha 22 da Figura 4-9, a execução da função *kernel* HistoryNeutronGPU, recebendo como parâmetro o novo número de histórias (plan→dataN).

```
1  /*********************************************
2   TRECHO CÓDIGO DisparaThreads
3  *********************************************/
4  static CUT_THREADPROC DisparaThreads(TGPUplan *plan )
5  {
6    cudaSetDevice(plan->device);
7    TpMeio          Meio;
8    unsigned long long  n, *count_h, totcount;
9    unsigned long long *count_d;
10
11   // Inicializando variáveis do host
12   totcount = 0;
13   count_h = (unsigned long long*) malloc(sizeof(unsigned long long) * NE);
14   memset(count_h, 0, sizeof(unsigned long long) * NE);
15   cudaMalloc((void **) &count_d, sizeof(unsigned long long) * NE);
16   cudaMemcpy(count_d, count_h, sizeof(unsigned long long) * NE, cudaMemcpyHostToDevice);
17
18   // inicializa dados físicos do meio
19   :
20
21   // Dispara o kernel
22   historyneutron_gpu<<< N_BLOCKS, N_THREADS >>>( rand(), Meio, count_d, plan->dataN);
23
24   cudaThreadSynchronize();
25
26   cudaMemcpy(count_h, count_d, sizeof(unsigned long long) * NE, cudaMemcpyDeviceToHost);
27   for (n = 0; n < NE; n++) totcount += count_h[n];
28   *(plan->h_Sum) = totcount;
29
30   //Shut down this GPU
31   free(count_h);
32   cudaFree(count_d);
33
34   CUT_THREADEND;
35 }
```

Figura 4-9 – Pseudocódigo da função DisparaKernel

4.4 Solução multi-GPU

O último passo deste trabalho constituiu-se em construir uma versão paralela do programa de simulação que rodasse num *cluster* de GPUs. Na solução apresentada nas Figuras 4-10, 4-11 e 4-12, usou-se o protocolo de comunicação síncrona através de MPI_Send e MPI_Recv.

```
/**********************************
  PSEUDOCODIGO DA SOLUÇÃO MULTI-GPU
**********************************/
int master();
int slave();
int ntasks, taskid;
unsigned long long buffsize;

int main(int argc,char** argv)
{
  // MPI Inicialização do MPI
  // Presente na lista de argumentos:  Meio, nro de nós, nro de blocos, nro de threads
  MPI_Init(&argc, &argv);

  // Inicializa o taskid ( ID desta tarefa) e ntasks (nro de nós )
  MPI_Comm_rank ( MPI_COMM_WORLD, &taskid );
  MPI_Comm_size ( MPI_COMM_WORLD, &ntasks );

  buffsize = NH / (ntasks - 1);

  if ( taskid==MASTER )
          master();
  else
          slave();

  // Finaliza MPI.                          */
  MPI_Finalize();
}
```

Figura 4-10 – Trecho programa principal da solução multi-GPU

Utilizando-se bibliotecas CUDA e MPI, foi desenvolvida uma solução, de forma que esta pudesse ser parametrizada quanto à alocação dos recursos computacionais. Assim, o número de nós no *cluster*, a quantidade de GPUs, bem

como a alocação de blocos e *threads* (parâmetros de configuração das GPUs) podem ser configuráveis.

Usando-se as funções MPI_Comm_rank(MPI_COMM_WORLD, &rank) e MPI_Comm_size (MPI_COMM_WORLD, &n_processos), pode-se controlar qual função será executada pelas máquinas *master* e *slave*, e quantos processos serão executados pelo experimento, respectivamente. A seguir, apresenta-se na Figura 4-11 o código *master*.

```
1  //************************* MASTER ******************************************
2
3  void master(int rank, long longhistorias)
4  {
5    int n_processos, i;
6    long long int total = 0, retorno = 0;
7    long long int historias_por_slave[n_processos];
8    MPI_Status status;
9
10   MPI_Comm_size(MPI_COMM_WORLD, &n_processos);
11
12   // Dividindo as histórias pelo número de processos a serem executados
13   for ( i = 0; i <n_processos; i++ )
14         historias_por_slave[i] = historias / (n_processos - 1);
15
16   // Caso a divisão aí de cima não tenha sido exata, ajusta as histórias.
17   for ( i = 0; i < historias % n_processos; i++ )
18        historias_por_slave[i]++;
19
20   // Distribuindo as tarefas pelos devices
21   for ( i = 1; i <n_processos; i++ ) {
22        MPI_Send(&historias_por_slave[i], 1, MPI_LONG_LONG_INT, i, 1, MPI_COMM_WORLD);
23   }
24
25   // Recebendo o total computado por cada device
26   for ( i = 1; i <n_processos; i++ ) {
27        MPI_Recv(&retorno, 1, MPI_LONG_LONG_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
28                      MPI_COMM_WORLD, &status);
29        total += retorno;
30   }
31
32   printf("%.8f;", (float) total / DATA_N);
33 }
```

Figura 4-11 – Trecho do código executado pelo *master*

Observe-se também na Figura 4-11, linhas 13 e 14, a divisão do número de histórias pelo número de processos MPI, enquanto a linha 22 distribui o número de histórias para cada *slave*, que será recebido pela função equivalente na linha 11 da Figura 4-12.

O laço compreendido entre as linhas 26 e 30, inclusive, da Figura 4-11, recebe a totalização de cada *slave* enviada pela linha 16 da Figura 4-12 e totaliza o número de histórias de cada nêutron do experimento.

```
1  //************************* SLAVE *******************************************
2
3  void slave(int rank, const char *elemento)
4  {
5    long long int n, resultado, numero_historias;
6    int numero_devices;
7    int total_threads = N_BLOCKS * N_THREADS;
8
9    // Configurando ambiente...
10   MPI_Status status;
11   MPI_Recv(&numero_historias, 1, MPI_LONG_LONG_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
12               MPI_COMM_WORLD, &status);
13   launch_montecarlo(rank, numero_historias, &resultado, elemento, N_BLOCKS, N_THREADS);
14
15   // Retornando os resultados para o master! master!
16   MPI_Send(&resultado, 1, MPI_LONG_LONG_INT, MASTER, 0, MPI_COMM_WORLD);
17
18 }
```

Figura 4-12 – Trecho de código a ser executado por cada *slave*

5 EXPERIMENTOS E RESULTADOS

Os experimentos realizados têm o objetivo de investigar quantitativamente os ganhos obtidos com a utilização do *cluster* de GPUs, assim como eventuais limitações, para o caso da simulação de transporte de nêutrons através do método Monte Carlo.

Foram considerados três meios distintos: i) água, ii) alumínio e iii) cádmio. Para cada um deles, foram simuladas 10^{10} histórias (equivalente ao número de nêutrons), de forma que o tempo total das simulações fosse da ordem de segundos (entre 2 e 60 segundos, aproximadamente), quando executados em uma GPU.

A versão paralela do programa desenvolvido foi executada considerando-se diversas configurações e variando-se os recursos computacionais alocados. Exaustivos experimentos, variando-se a quantidade de nós do *cluster*, o número de GPUs, bem como o número de blocos e *threads* (parâmetros de configuração das GPUs) foram executados, visando-se obter configurações otimizadas e medir os ganhos e limitações do sistema desenvolvido. Neste trabalho, utilizaram-se até oito GPUs. Para efeito de comparação, uma versão sequencial (não paralela) da mesma simulação foi desenvolvida.

5.1 Experimentos com 1 GPU

Os primeiros testes foram realizados utilizando-se apenas uma GPU. A transferência de dados da memória do *host* para a memória do *device* (CPU-GPU) foi realizada através de chamadas cudaMemcpy. Estes experimentos visaram, além de checar o resultado dos cálculos executados, um estudo do comportamento (sensibilidade) dos tempos de processamento com relação à alocação de blocos e *threads*.

Como condições de contorno dadas pelas especificações da GPU utilizada[32] e recomendações conforme a *compute capability* da GTX480, adotou-se:

---

[32] Disponível em: http://www.geforce.com/Hardware/GPUs/geforce-gtx-480/specifications. Acessado em: 29 fev. 2012.
NVIDIA GeForce GTX 480/470/465 GPUDatasheet, 2010. Cf., também, NVIDIA CUDA C Programming Guide, Version 3.2, 22/10/2010, p. 111.

- o número máximo de *threads* por bloco = 1.024
- o número de blocos = múltiplo de 15

Adotou-se o valor 15 não por acaso para o número de blocos; isto se deu em razão de a GTX480 possuir 15 SMs. A quantidade total de processos (total de *threads*) enviados para execução na GPU (na função *kernel*) é dada pelo produto entre o número de blocos e o número de *threads* por bloco:

$$\text{total\_}threads = \text{blocos} \times threads \tag{5.1}$$

Onde:

- total_*threads* = número total de *threads* alocados para execução de uma função de *kernel*
- blocos = número de blocos
- *threads* = número de *threads* por bloco

Assim sendo, dividiram-se os experimentos em grupos onde total_*threads* era fixo, enquanto blocos e *threads* eram variáveis. As Tabelas 5-1 a 5-3, bem como os Gráficos 5-1 a 5-3, exibem os resultados destes experimentos. Os valores de tempos exibidos nestas tabelas são médias de três experimentos (execuções). Consideraram-se apenas três experimentos devido ao fato de a variação entre as medidas de tempo entre os experimentos ser menor do que 1%.

Tabela 5-1 – Resultados obtidos 1 GPU. Meio: ÁGUA

Meio: ÁGUA			**Fator de transmissão (teórico): 0,8025**	
Blocos x *Threads*	**Blocos**	***Threads***	**Tempo (s)**	**Fator**
122.880	3.840	32	84,72	0,8025
	1.920	64	60,78	0,8025
	960	128	59,97	0,8025
	480	256	59,91	0,8025
	240	512	59,84	0,8025
	120	1.024	59,84	0,8025
61.440	1.920	32	85,32	0,8025
	960	64	62,02	0,8025
	480	128	59,94	0,8025
	240	256	59,84	0,8025
	120	512	59,85	0,8025
	60	1.024	59,82	0,8025
30.720	960	32	85,84	0,8025
	480	64	62,75	0,8025
	240	128	59,94	0,8025
	120	256	59,95	0,8025
	60	512	59,84	0,8025
	30	1.024	59,80	0,8025
15.360	480	32	59,85	0,8025
	240	64	59,82	0,8025
	120	128	59,95	0,8025
	60	256	59,98	0,8025
	30	512	59,87	0,8025
	15	1.024	59,80	0,8025

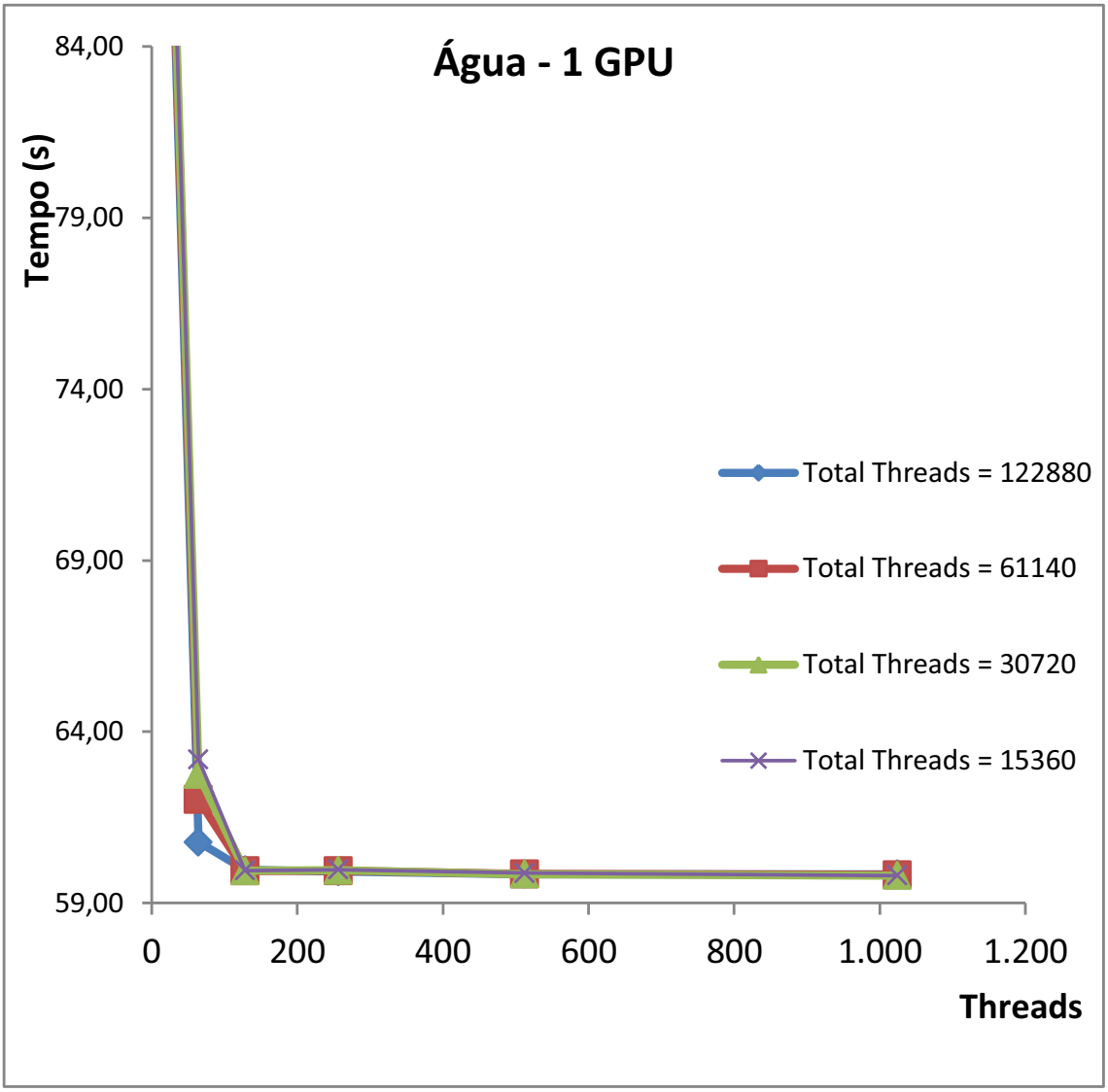

Gráfico 5-1 – Resultados obtidos utilizando 1 GPU. Meio: ÁGUA

Tabela 5-2 − Resultados obtidos utilizando 1 GPU. Meio: ALUMÍNIO

Meio: ALUMÍNIO			**Fator de transmissão (teórico): 0,8607**	
Blocos x *Threads*	**Blocos**	***Threads***	**Tempo (s)**	**Fator**
122.880	3.840	32	8,51	0,8607
	1.920	64	6,34	0,8607
	960	128	6,27	0,8607
	480	256	6,26	0,8607
	240	512	6,27	0,8607
	120	1.024	6,29	0,8607
61.440	1.920	32	8,53	0,8607
	960	64	6,38	0,8607
	480	128	6,26	0,8607
	240	256	6,26	0,8607
	120	512	6,26	0,8607
	60	1.024	6,27	0,8607
30.720	960	32	8,55	0,8607
	480	64	6,48	0,8607
	240	128	6,25	0,8607
	120	256	6,25	0,8607
	60	512	6,25	0,8607
	30	1.024	6,19	0,8607
15.360	480	32	6,26	0,8607
	240	64	6,27	0,8607
	120	128	6,24	0,8607
	60	256	6,25	0,8607
	30	512	6,25	0,8607
	15	1.024	6,25	0,8607

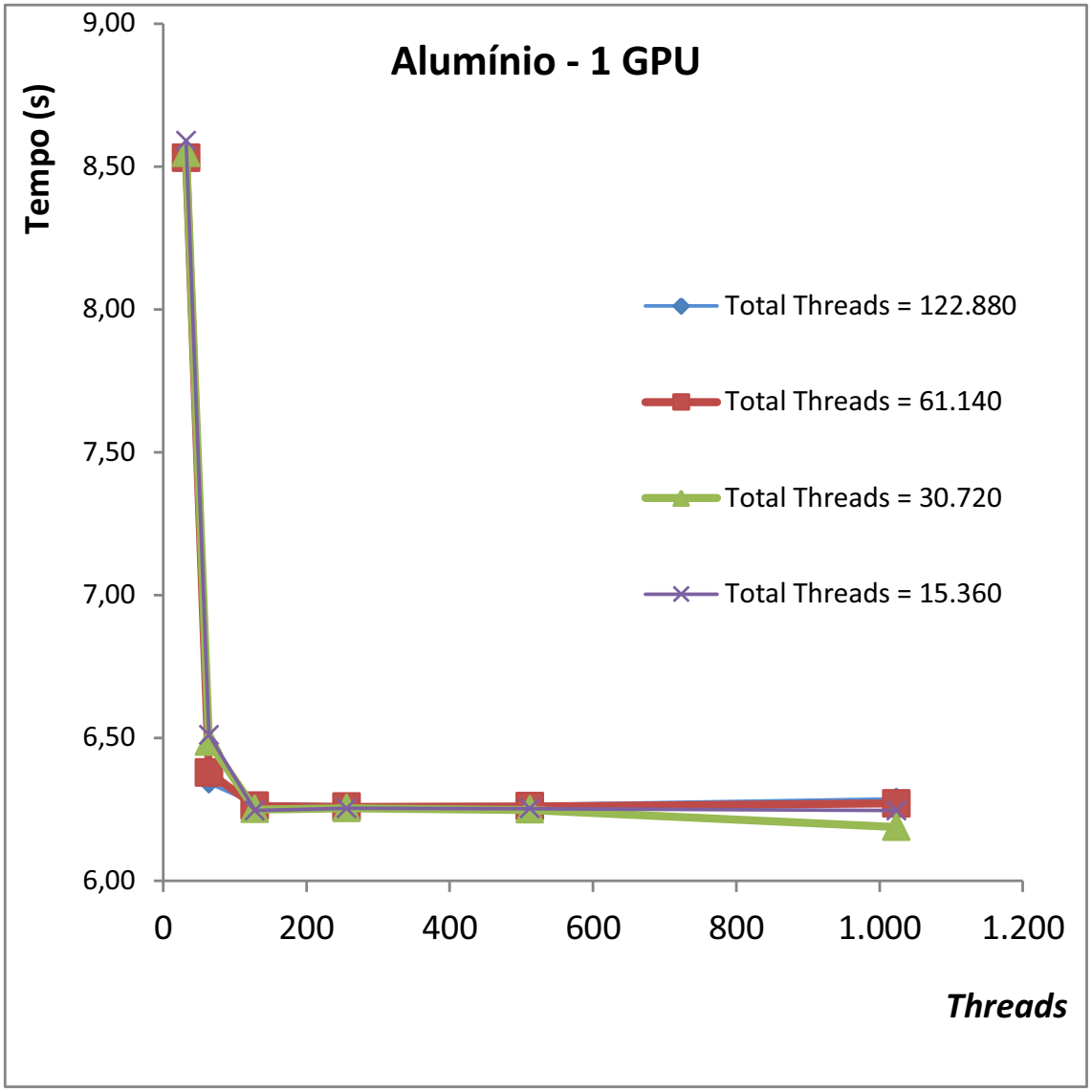

Gráfico 5-2 – Resultados obtidos utilizando 1 GPU. Meio: ALUMÍNIO

Tabela 5-3 − Resultados obtidos utilizando 1 GPU. Meio: CÁDMIO

Meio: CÁDMIO		**Fator de transmissão (teórico): 0,3198**		
Blocos x *Threads*	**Blocos**	***Threads***	**Tempo (s)**	**Fator**
122.880	3.840	32	3,38	0,3198
	1.920	64	2,72	0,3198
	960	128	2,70	0,3198
	480	256	2,71	0,3198
	240	512	2,70	0,3198
	120	1.024	2,72	0,3198
61.440	1.920	32	3,35	0,3198
	960	64	2,68	0,3198
	480	128	2,69	0,3198
	240	256	2,68	0,3198
	120	512	2,68	0,3198
	60	1.024	2,69	0,3198
30.720	960	32	3,35	0,3198
	480	64	2,74	0,3198
	240	128	2,67	0,3198
	120	256	2,68	0,3198
	60	512	2,68	0,3198
	30	1.024	2,68	0,3198
15.360	480	32	2,68	0,3198
	240	64	2,69	0,3198
	120	128	2,68	0,3198
	60	256	2,67	0,3198
	30	512	2,67	0,3198
	15	1.024	2,66	0,3198

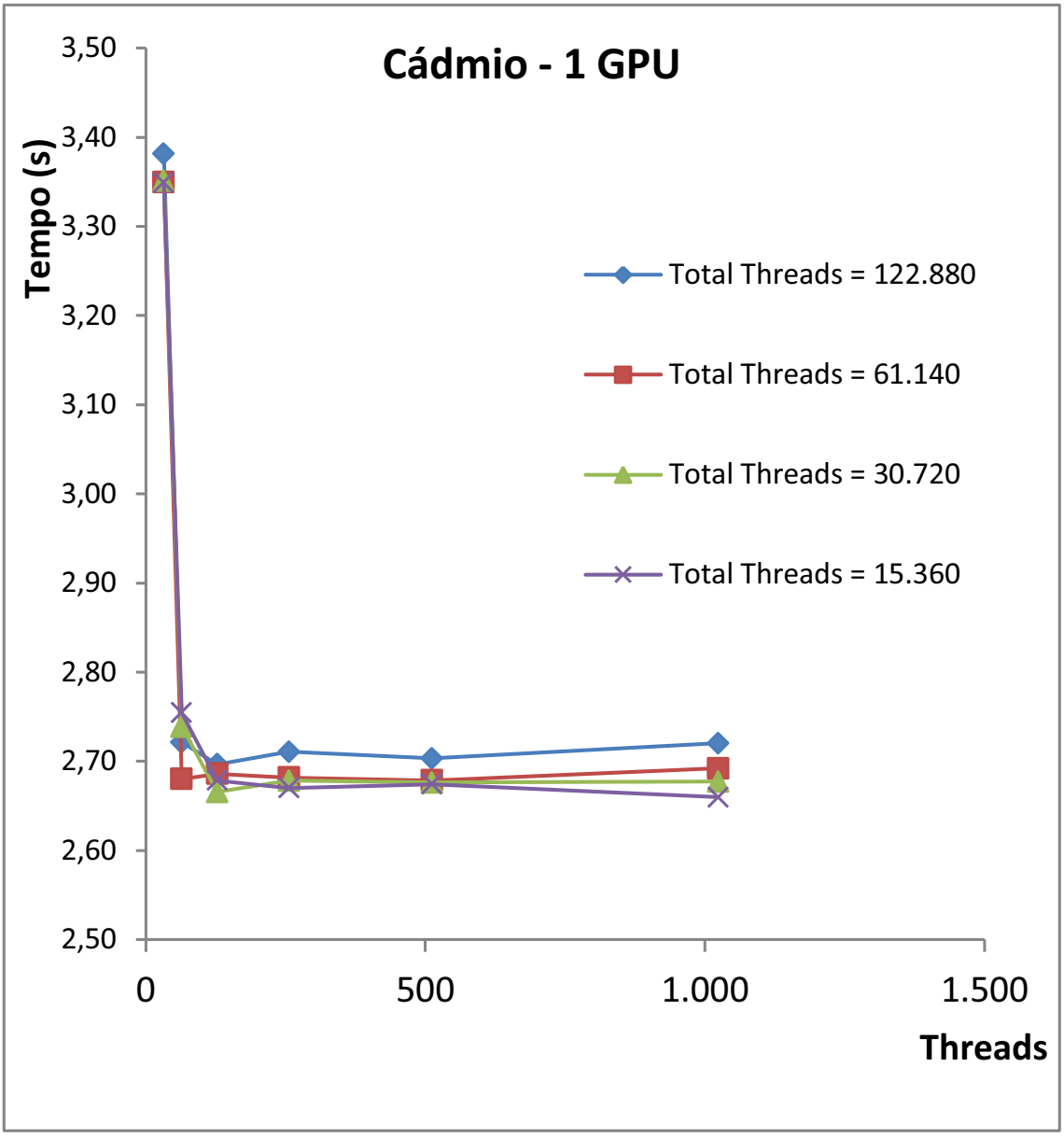

Gráfico 5-3 – Resultados obtidos utilizando 1 GPU. Meio: CÁDMIO

Inicialmente, verificaram-se os valores obtidos para o fator de transmissão. O que se observou foi que, em todos os casos, o fator de transmissão convergia para aproximadamente o mesmo valor (exceto variações na quinta ou sexta casa decimal) e aproximavam-se satisfatoriamente os valores teóricos.

Os Gráficos 5-1 a 5-3 demonstram uma tendência de convergência dos tempos de execução para as redondezas de um certo valor de tempo, que fica aproximadamente assintótico ao eixo do número total de *threads* (total_*threads*). Assim sendo, buscando ratificar esta observação, continuou-se a investigar o comportamento para uma quantidade ainda menor de total_*threads*, diminuindo-se ainda mais a quantidade de blocos. As Tabelas 5-4 a 5-9 exibem estes resultados.

Constatou-se também que a variação do número total de *threads* (blocos x *threads*) não acarretou mudanças significativas nos tempos de execução; entretanto, números de blocos elevados implicaram um aumento no custo computacional. Para os três meios, a utilização de um número reduzido de blocos com um número elevado de *threads* por bloco gerou simulações mais rápidas.

Por outro lado, um número reduzido de *threads* também parece piorar os tempos. Para averiguar estas variáveis (blocos e *threads*), foram feitos experimentos, fixando-se cada um deles.

Tabela 5-4 − Experimentos com a quantidade de *threads* fixos. Meio: ÁGUA

Água – 1 GPU				
Blocos x *Threads*	**Blocos**	***Threads***	**Tempo (s)**	**Fator**
122.880	3.840	32	84,72	0,8025
61.440	1.920	32	85,32	0,8025
30.720	960	32	85,34	0,8025
15.360	480	32	82,18	0,8025

Tabela 5-5 – Experimentos com a quantidade de blocos fixos. Meio: ÁGUA

Água – 1 GPU				
Blocos x Threads	**Blocos**	***Threads***	**Tempo (s)**	**Fator**
122.880	3.840	32	84,72	0,8025
245.760	3.840	64	60,32	0,8025
491.520	3.840	128	60,56	0,8026
983.040	3.840	256	62,25	0,8026
196.6080	3.840	512	65,55	0,8026

Tabela 5-6 – Experimentos com a quantidade de *threads* fixos. Meio: ALUMÍNIO

Alumínio – 1 GPU				
Blocos x *Threads*	**Blocos**	***Threads***	**Tempo (s)**	**Fator**
122.880	3.840	32	8,54	0,8607
61.440	1.920	32	8,53	0,8607
30.720	960	32	8,54	0,8607
15.360	480	32	8,59	0,8607

Tabela 5.7 – Experimentos com a quantidade de blocos fixos. Meio: ALUMÍNIO

Alumínio – 1 GPU				
Blocos x *Threads*	**Blocos**	***Threads***	**Tempo (s)**	**Fator**
122.880	3.840	32	8,54	0,8607
245.760	3.840	64	6,29	0,8607
491.520	3.840	128	7,00	0,8608
983.040	3.840	256	8,70	0,8608
1.966.080	3.840	512	15,11	0,8608

Tabela 5-8 – Experimentos com a quantidade de *threads* fixos. Meio: CÁDMIO

Cádmio – 1 GPU				
Blocos x *Threads*	**Blocos**	***Threads***	**Tempo (s)**	**Fator**
122.880	3.840	32	3,38	0,3198
61.440	1.920	32	3,35	0,3198
30.720	960	32	3,35	0,3198
15.360	480	32	3,35	0,3199

Tabela 5-9 – Experimentos com a quantidade de blocos fixos. Meio: CÁDMIO

Cádmio – 1 GPU				
Blocos x *Threads*	**Blocos**	***Threads***	**Tempo (s)**	**Fator**
122.880	3.840	32	3,38	0,3198
245.760	3.840	64	2,77	0,3198
491.520	3.840	128	3,70	0,3198
983.040	3.840	256	7,02	0,3198
1.966.080	3.840	512	14,25	0,3198

Um fato visível nestes últimos experimentos é que um produto elevado de Blocos x *threads* (exemplo: 983.040 e 1.966.080) cria problemas na execução da GPU que parece não conseguir gerenciá-los de forma eficiente.

Embora haja uma considerável não linearidade no comportamento do tempo de execução quando variados Blocos e threads, observou-se que a combinação de número alto de blocos e número baixo de *threads*, 3.840x64, obteve os menores dos tempos, e que estes tempos ainda são maiores do que o obtido para a combinação de 15x1.024 de todos os elementos.

5.2 Experimentos com 2 GPUs

Os mesmos procedimentos, variando-se as configurações de blocos e *threads*, foram feitos para duas GPUs em um mesmo computador, sendo cada uma delas controlada por processos independentes, em núcleos (*cores*) independentes. Os resultados obtidos ratificam as constatações feitas para uma GPU, ou seja, a *performance* em termos de tempo de processamento é diminuída para uma quantidade mais alta de blocos, que fica aproximadamente assintótico ao eixo do número total de *threads* (total_*threads*).

As tabelas 5-10 a 5-12 exibem os resultados das simulações realizadas utilizando-se 2 GPUs.

Tabela 5-10 – Resultados obtidos utilizando 2 GPUs. Meio: ÁGUA

Meio: ÁGUA			**Fator de transmissão (teórico): 0,8025**	
Blocos x *Threads*	**Blocos**	***Threads***	**Tempo(s)**	**Fator**
122.880	3.840	32	43,06	0,8025
	1.920	64	31,07	0,8025
	960	128	30,65	0,8025
	480	256	30,64	0,8025
	240	512	30,59	0,8025
	120	1.024	30,6	0,8025
61.440	1920	32	43,34	0,8025
	960	64	62,02	0,8025
	480	128	30,64	0,8025
	240	256	30,64	0,8025
	120	512	30,58	0,8025
	60	1.024	30,58	0,8025
30.720	960	32	43,59	0,8025
	480	64	32,02	0,8025
	240	128	30,63	0,8025
	120	256	30,63	0,8025
	60	512	30,6	0,8025
	30	1.024	30,56	0,8025
15.360	480	32	43,77	0,8025
	240	64	32,26	0,8025
	120	128	30,63	0,8025
	60	256	30,65	0,8025
	30	512	30,59	0,8025
	15	1.024	30,56	0,8025

Tabela 5-11 – Resultados obtidos utilizando 2 GPUs. Meio: ALUMÍNIO

Meio: ALUMÍNIO			**Fator de transmissão (teórico): 0,8607**	
Blocos x *Threads*	**Blocos**	***Threads***	**Tempo (s)**	**Fator**
122.880	3.840	32	4,94	0,8607
	1.920	64	3,85	0,8607
	960	128	3,80	0,8607
	480	256	3,81	0,8607
	240	512	3,82	0,8607
	120	1.024	3,82	0,8607
61.440	1.920	32	4,94	0,8607
	960	64	3,83	0,8607
	480	128	3,80	0,8607
	240	256	3,80	0,8607
	120	512	3,79	0,8607
	60	1.024	3,80	0,8607
30.720	960	32	4,95	0,8607
	480	64	3,90	0,8607
	240	128	3,79	0,8607
	120	256	3,79	0,8607
	60	512	3,79	0,8607
	30	1.024	3,73	0,8607
15.360	480	32	4,96	0,8607
	240	64	3,92	0,8607
	120	128	3,78	0,8607
	60	256	3,79	0,8607
	30	512	3,79	0,8607
	15	1.024	3,78	0,8607

Tabela 5-12 – Resultados obtidos utilizando 2 GPUs: Meio: CÁDMIO

Meio: CÁDMIO			**Fator de transmissão (teórico): 0,3198**	
Blocos x *Threads*	**Blocos**	***Threads***	**Tempo(s)**	**Fator**
122.880	3.840	32	2,37	0,3198
	1.920	64	2,03	0,3198
	960	128	2,01	0,3198
	480	256	2,03	0,3198
	240	512	2,03	0,3198
	120	1.024	2,04	0,3198
61.440	1.920	32	2,34	0,3198
	960	64	1,98	0,3198
	480	128	2,01	0,3198
	240	256	2,01	0,3198
	120	512	2,02	0,3198
	60	1.024	2,01	0,3198
30.720	960	32	2,33	0,3198
	480	64	2,03	0,3198
	240	128	1,99	0,3198
	120	256	2,00	0,3198
	60	512	2,00	0,3198
	30	1.024	2,00	0,3198
15.360	480	32	2,34	0,3198
	240	64	2,04	0,3198
	120	128	2,00	0,3198
	60	256	2,00	0,3198
	30	512	2,00	0,3198
	15	1.024	1,99	0,3198

Nestes experimentos, observou-se uma diferença constante de aproximadamente 0,6 segundo nos valores de tempos obtidos para todos os materiais, onde se esperavam valores que corresponderiam a um valor próximo da metade dos valores para uma GPU. O fato de esta diferença ser constante acarreta

pequeno impacto relativo em problemas de alta demanda computacional; por exemplo, um problema que levasse 10 horas para ser executado em uma GPU poderia ser executado em 5 horas mais 0,6 segundo em duas GPUs, o que é uma diferença relativamente pequena. Entretanto, poderia apontar para uma restrição no caso de problemas cuja execução em 1 GPU já fosse da ordem de 1 segundo. Neste caso, a execução em 2 GPUs levaria 1,1 segundo (0,5+0,6).

Inicialmente, creditou-se esta diferença a basicamente a dois fatores: i) comunicação entre CPUs e GPUs (*host-device* e *device-host*) e ii) acréscimo referente à parte de processamento sequencial (incluindo o gerenciamento de *threads* na CPU).

Investigando-se um pouco mais sobre *performance* de multi-GPUs, descobriu-se que a NVIDIA disponibiliza uma API baseada em C para Linux, que faz o monitoramento e gerenciamento dos modos de configuração de todas as GPUs instaladas.[33] A detecção do *hardware* presente pelo programa é mostrada na Figura 5-1.

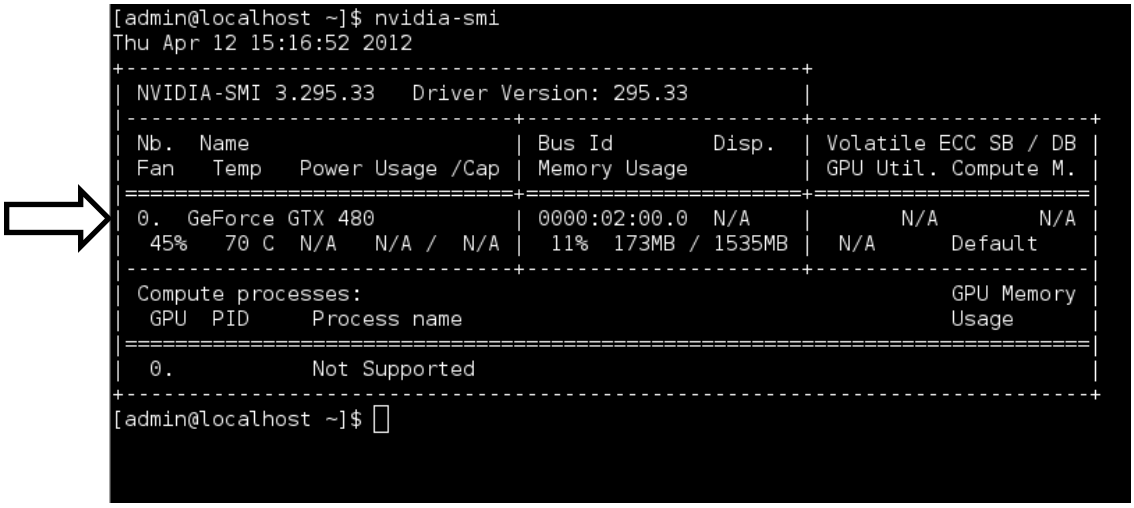

Figura 5-1 – Tela exibida após execução do programa nvidia-smi

Um desses modos, conhecido como *persistence mode*, controla se o *driver* NVIDIA permanece carregado quando há clientes ativos ligados à GPU. Conforme se observa na Figura 5-2, a GPU GTX480 usada neste experimento não está na lista

[33] Disponível em http://developer.nvidia.com/nvidia-management-library-nvml. Acessado em: 08 mar. 2012.

de produtos suportados, porém, ao se habilitar o modo *persistence mode* através do programa nvidia-smi, observou-se que o mesmo foi ativado.

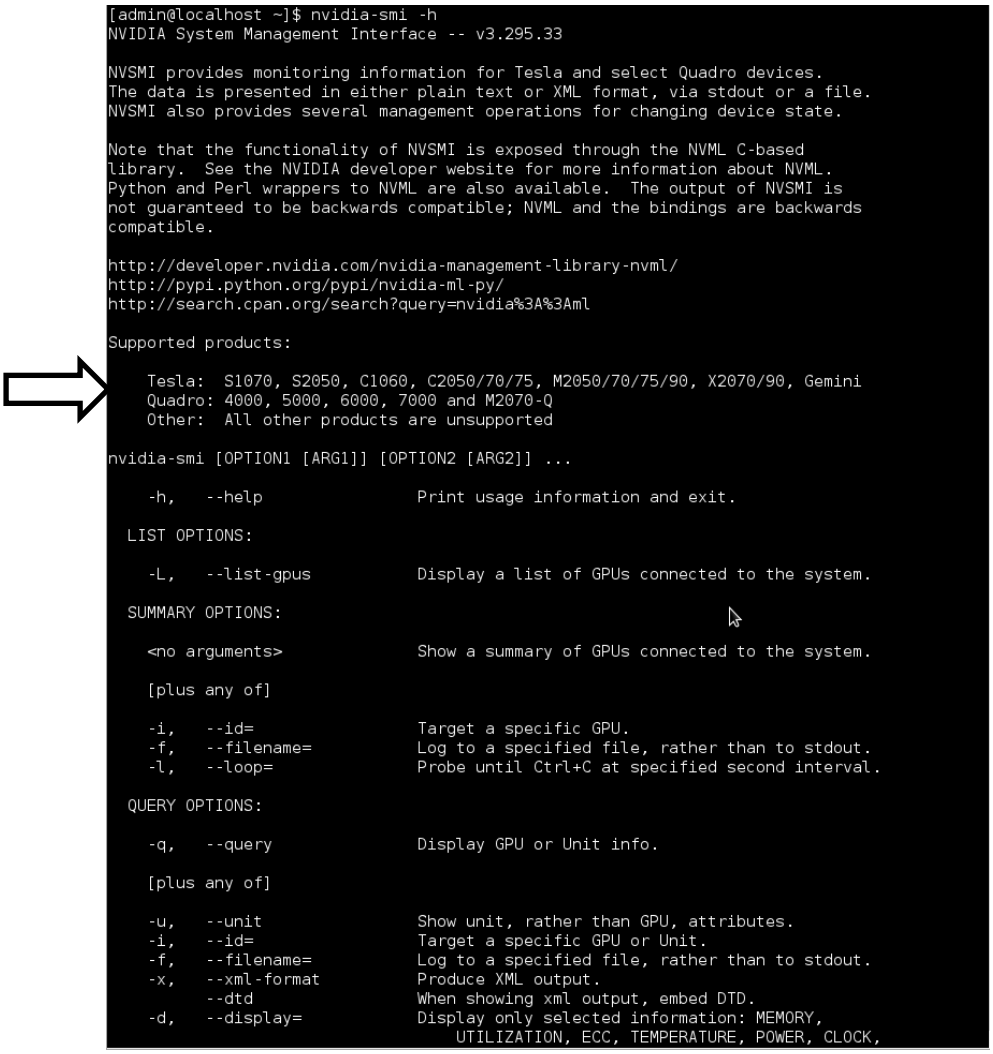

Figura 5-2 – *States* controlado pelo programa nvidia-smi

Ao se executar todo o experimento com o modo *persistence mode* ligado, observou-se uma relevante melhoria na *performance* em todos os meios. O acréscimo de 0,6s caiu para aproximadamente 0,15 segundos. Entretanto, observou-se uma considerável margem de erros, intermitentes que geravam fatores de transmissão de nêutrons incoerentes, parecendo algum mal funcionamento do *hardware* com este modo de configuração. Ficou patente que o fato do *driver* ficar sempre carregado economiza tempo e que pode ser precioso em simulações um pouco mais rápidas, entretanto, como o *hardware* que utilizamos não suportaria

(supostamente) esta função, resolveu-se considera-la em futuras investigações (fora desta dissertação).

5.3 Experimentos com mais de 2 GPUs (multi-GPUs)

A justificativa para se utilizarem multi-GPUs poderia ser o elevado volume de dados. Por exemplo: em White Paper 3D Finite Difference Computation on GPUs Using CUDA, o volume de dados alcançou mais até 10 GB para serem processados de uma única vez numa solução paralela em GPU Tesla S1070 com capacidade de 4 GB. Por isso, fez-se necessário o uso de mais GPUs, para particionar o enorme volume de dados em multi-GPUs. Diferentemente, o uso de multi-GPUs neste trabalho teve dois motivos. Primeiro, objetivou obter o ganho em *speedup*[34] (que pode ser obtido em relação à solução sequencial para o cálculo de 10^{10} histórias de nêutrons a atravessar um *slab*). Segundo, que é consequência do primeiro, adquirir conhecimento para o desenvolvimento de um problema de alto custo computacional da área nuclear.

Para se usarem múltiplos contextos CUDA, pode-se associá-los aos *threads* de diferentes *cores*, um para cada GPU. Para um desempenho ideal, o número de *cores* de CPU não deve ser menor do que o número de GPUs no sistema (SPAMPINATO; ELSTER, s.d.). O gerenciamento dos *threads* pode ser feito por implementação de uma camada de comunicação, presente no sistema operacional Linux, através de bibliotecas do sistema, como a *Posix Threads*, usada por Almeida (2009). Neste trabalho, adotou-se o MPI para realização da comunicação entre os diversos nós do *cluster*.

A solução paralela foi executada em um número variável de GPUs, onde as 10^{10} histórias de nêutrons foram divididas pelo número de nós e de GPUs de cada experimento. Os mesmos procedimentos, variando-se as configurações de blocos e *threads*, foram feitos para multi-GPUs.

Objetivamente, buscou-se compreender o comportamento da *performance* com o mesmo experimento nas diferentes máquinas que compõem o *cluster*. Partindo do melhor resultado de configuração do número de blocos x *threads*

[34] *Speedup* é a relação entre o tempo de execução sequencial e o tempo de execução paralela. QUINN. Op. cit., Capítulo 7, p. 160.

definida nos ensaios com duas GPUs, encontraram-se os resultados das Tabelas e dos Gráficos a seguir.

Meio: Água

Fator teórico **0,8025**

Tempo sequencial(seg) **18.281,57**

Nro total de *threads*	Blocos	*Threads*	1 nó				2 nós		3 nós		4 nós	
			1 GPU		**2 GPU's**		**4 GPU's**		**6 GPU's**		**8 GPU's**	
			Fator	**t(s)**	**Fator**	**t(s)**	**Fator**	**t(s)**	**Fator**	**t(s)**	**Fator**	**t(s)**
1920	15	128	0,802520	169,19	0,802517	85,25	0,802522	43,27	0,802515	29,27	0,802522	22,27
3840	15	256	0,802520	123,08	0,802521	62,20	0,802518	31,74	0,802519	21,60	0,802515	16,52
4800	15	320	0,802519	119,89	0,802520	60,62	0,802520	30,94	0,802522	21,05	0,802521	16,12
9600	15	640	0,802520	60,11	0,802521	30,73	0,802518	15,99	0,802526	11,09	0,802522	8,70
15360	480	32	0,80252	86,18	0,80252	43,77	0,80252	22,52	0,80252	15,44	0,80252	11,93
15360	240	64	0,80252	63,20	0,80252	32,26	0,80252	16,77	0,80252	11,61	0,80253	9,05
15360	120	128	0,80252	59,95	0,80252	30,63	0,80252	15,96	0,80252	11,07	0,80252	8,64
15360	60	256	0,80252	59,98	0,80252	30,65	0,80252	15,96	0,80252	11,06	0,80252	8,64
15360	30	512	0,80252	59,87	0,80252	30,59	0,80252	15,94	0,80252	11,05	0,80252	8,61
15360	15	1024	0,80252	59,80	0,80252	30,56	0,80252	15,91	0,80252	11,05	0,80252	8,62
30720	960	32	0,80252	85,84	0,80253	43,59	0,80253	22,43	0,80253	15,40	0,80254	11,85
30720	480	64	0,80252	62,75	0,80252	32,03	0,80253	16,65	0,80252	11,52	0,80254	8,96
30720	240	128	0,80252	59,94	0,80252	30,63	0,80253	15,96	0,80253	11,06	0,80254	8,62
30720	120	256	0,80252	59,95	0,80252	30,63	0,80253	16,05	0,80253	11,07	0,80254	8,62
30720	60	512	0,80252	59,84	0,80252	30,60	0,80252	15,93	0,80253	11,07	0,80254	8,63
30720	30	1024	0,80252	59,80	0,80252	30,56	0,80253	15,93	0,80253	11,06	0,80253	8,60
61440	1920	32	0,80252	85,32	0,80253	43,34	0,80253	22,32	0,80253	15,30	0,80255	11,82
61440	960	64	0,80252	62,02	0,80252	31,64	0,80254	16,49	0,80253	11,43	0,80256	8,88
61440	480	128	0,80253	59,94	0,80252	30,64	0,80253	15,97	0,80252	11,06	0,80256	8,62
61440	240	256	0,80252	59,94	0,80253	30,64	0,80254	15,97	0,80252	11,08	0,80255	8,62
61440	120	512	0,80253	59,85	0,80253	30,58	0,80254	15,93	0,80252	11,06	0,80256	8,63
61440	60	1024	0,80252	59,82	0,80252	30,58	0,80254	15,94	0,80253	11,08	0,80256	8,61
122880	3840	32	0,80253	84,72	0,80254	43,06	0,80256	22,17	0,80256	15,24	0,80256	11,77
122880	1920	64	0,80253	60,78	0,80253	31,07	0,80256	16,18	0,80256	11,23	0,80256	8,76
122880	960	128	0,80252	59,97	0,80254	30,65	0,80256	15,97	0,80256	11,08	0,80256	8,67
122880	480	256	0,80253	59,91	0,80254	30,64	0,80256	15,97	0,80256	11,10	0,80256	8,65
122880	240	512	0,80253	59,84	0,80254	30,59	0,80256	15,94	0,80255	11,06	0,80256	8,61
122880	120	1024	0,80253	59,84	0,80254	30,60	0,80256	15,97	0,80256	11,11	0,80256	8,64
245760	3840	64	0,80254	60,32	0,80256	30,83	0,80256	16,09	0,80256	11,20	0,80263	8,72
491520	3840	128	0,80256	60,56	0,80256	31,26	0,80264	16,71	0,80256	11,79	0,80280	9,40
983040	3840	256	0,80256	62,25	0,80264	32,97	0,80279	18,31	0,80279	13,43	0,80279	11,02
1966080	3840	512	0,80263	65,55	0,80279	36,61	0,80279	22,28	0,80279	17,85	0,80280	16,15

Tabela 5-13 – Resultados obtidos utilizando multiGPUs. Meio: ÁGUA

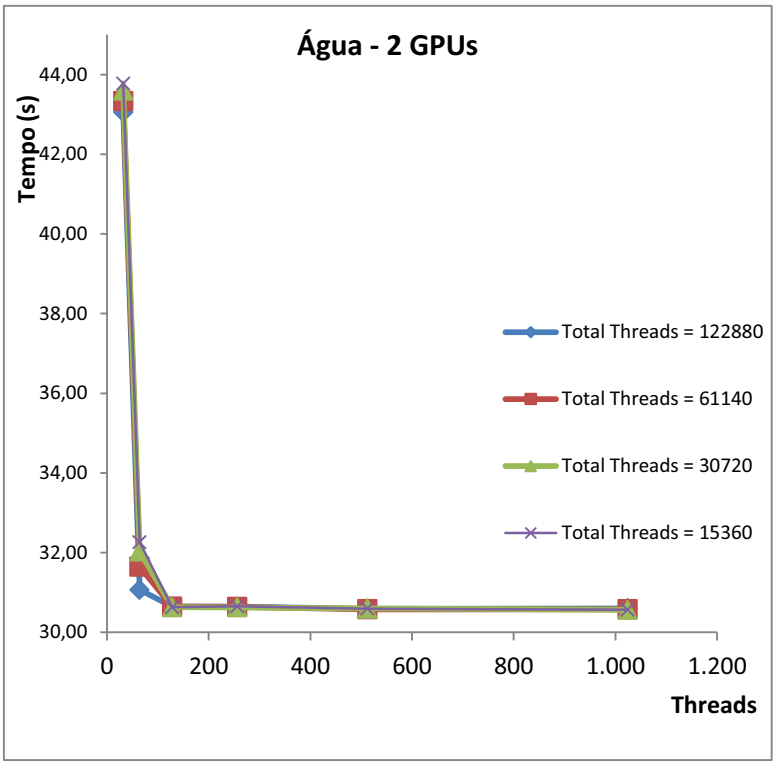

Gráfico 5-4 – Multi-GPUs (2 GPUs). Meio Água

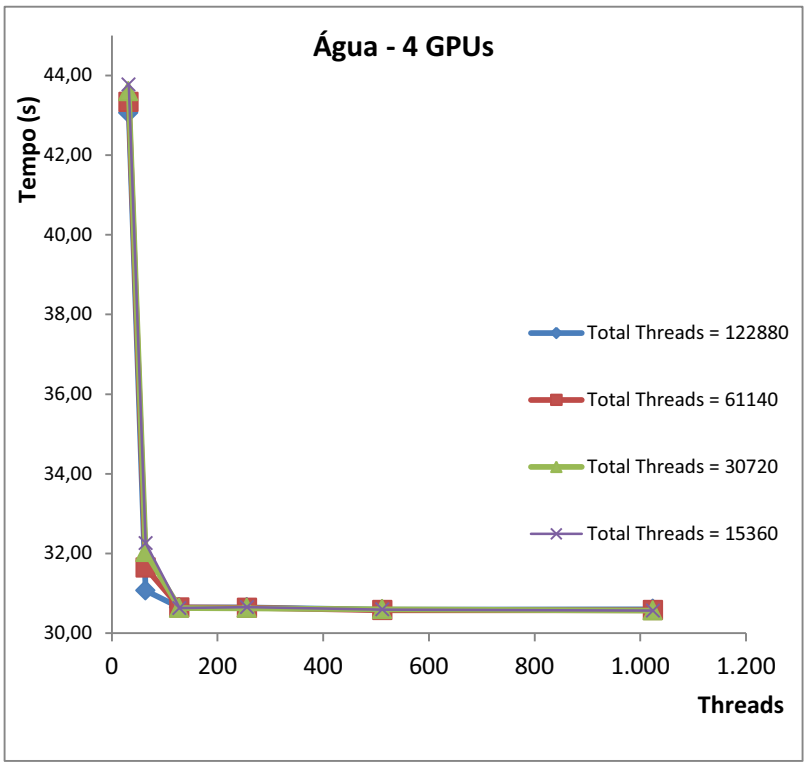

Gráfico 5-5 – Multi-GPUs (4 GPUs). Meio Água

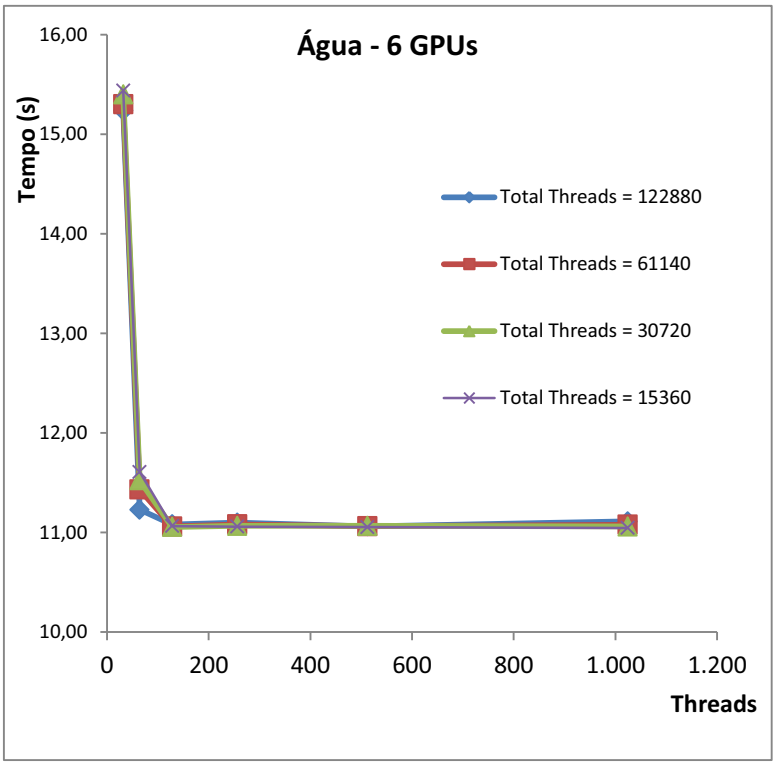

Gráfico 5-6 – Multi-GPUs (6 GPUs). Meio Água

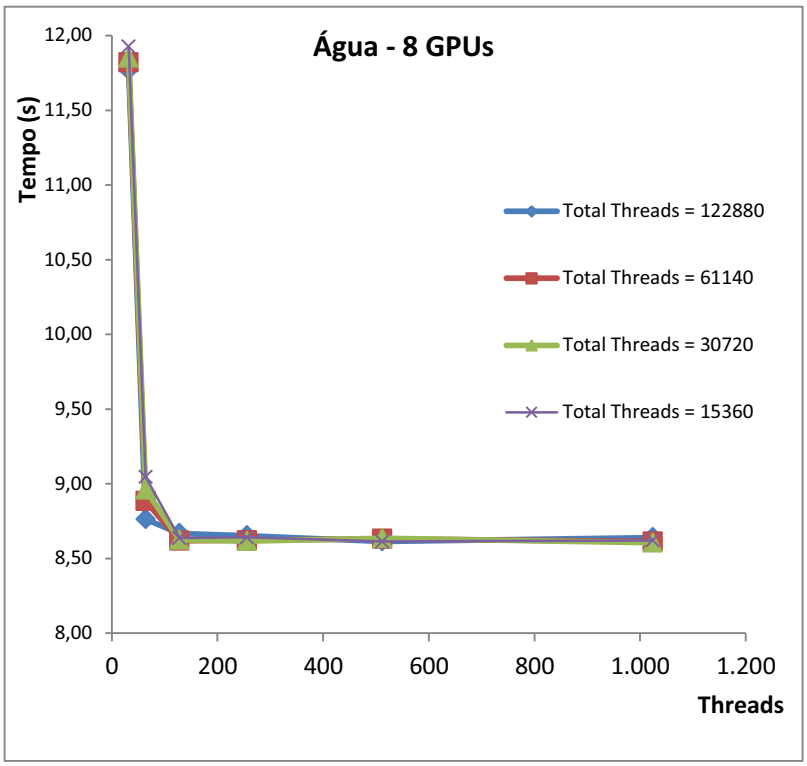

Gráfico 5-7 – Multi-GPUs (8 GPUs). Meio Água

Meio: Alumínio

Fator teórico **0,8607**

Tempo sequencial(seg) **1.166,48**

Nro total de *threads*	Blocos	*Threads*	1 nó				2 nós		3 nós		4 nós	
			1 GPU		2 GPU's		4 GPU's		6 GPU's		8 GPU's	
			Fator	t(s)	Fator	t(s)	Fator	t(s)	Fator	t(s)	Fator	t(s)
1920	15	128	0,86071	15,00	0,86071	7,76	0,86071	4,15	0,86071	2,94	0,86071	2,34
3840	15	256	0,86071	10,87	0,86071	5,69	0,86071	3,11	0,86071	2,25	0,86071	1,82
4800	15	320	0,86071	10,61	0,86071	5,57	0,86071	3,04	0,86071	2,20	0,86071	1,80
9600	15	640	0,86071	5,74	0,86071	3,21	0,86071	2,03	0,86072	1,61	0,86072	1,33
15360	480	32	0,86071	8,59	0,86071	4,96	0,86072	3,11	0,86071	2,51	0,86072	2,19
15360	240	64	0,86071	6,51	0,86071	3,92	0,86071	2,59	0,86072	2,15	0,86072	1,96
15360	120	128	0,86071	6,24	0,86071	3,78	0,86072	2,56	0,86072	2,11	0,86072	1,94
15360	60	256	0,86071	6,25	0,86071	3,79	0,86071	2,54	0,86071	2,14	0,86072	1,90
15360	30	512	0,86072	6,25	0,86071	3,79	0,86072	2,53	0,86071	2,12	0,86072	1,92
15360	15	1024	0,86071	6,25	0,86071	3,78	0,86071	2,55	0,86071	2,11	0,86072	1,94
30720	960	32	0,86071	8,55	0,86072	4,95	0,86072	3,11	0,86072	2,51	0,86073	2,19
30720	480	64	0,86072	6,48	0,86072	3,90	0,86072	2,59	0,86072	2,18	0,86073	1,95
30720	240	128	0,86071	6,25	0,86071	3,79	0,86072	2,54	0,86072	2,14	0,86073	1,91
30720	120	256	0,86071	6,25	0,86072	3,79	0,86072	2,63	0,86072	2,12	0,86073	1,91
30720	60	512	0,86071	6,25	0,86072	3,79	0,86072	2,53	0,86072	2,11	0,86073	1,93
30720	30	1024	0,86071	6,19	0,86072	3,73	0,86072	2,52	0,86072	2,10	0,86073	1,90
61440	1920	32	0,86072	8,53	0,86072	4,94	0,86073	3,11	0,86072	2,50	0,86075	2,21
61440	960	64	0,86071	6,38	0,86072	3,83	0,86073	2,56	0,86072	2,14	0,86076	1,92
61440	480	128	0,86072	6,26	0,86072	3,80	0,86073	2,55	0,86072	2,15	0,86076	1,92
61440	240	256	0,86072	6,26	0,86072	3,80	0,86073	2,55	0,86072	2,18	0,86075	1,91
61440	120	512	0,86071	6,26	0,86072	3,79	0,86073	2,54	0,86072	2,11	0,86075	1,92
61440	60	1024	0,86072	6,27	0,86072	3,80	0,86073	2,54	0,86072	2,13	0,86075	1,92
122880	3840	32	0,86072	8,51	0,86073	4,94	0,86076	3,13	0,86075	2,53	0,86075	2,26
122880	1920	64	0,86072	6,34	0,86073	3,85	0,86075	2,57	0,86075	2,15	0,86075	1,98
122880	960	128	0,86072	6,27	0,86073	3,80	0,86076	2,56	0,86075	2,13	0,86075	1,93
122880	480	256	0,86072	6,26	0,86073	3,81	0,86075	2,57	0,86075	2,16	0,86075	1,94
122880	240	512	0,86072	6,27	0,86073	3,82	0,86075	2,57	0,86075	2,16	0,86076	1,93
122880	120	1024	0,86072	6,29	0,86073	3,82	0,86076	2,58	0,86075	2,18	0,86075	1,95
245760	3840	64	0,86073	6,28	0,86075	3,82	0,86075	2,59	0,86075	2,23	0,86084	1,96
491520	3840	128	0,86075	7,00	0,86075	4,56	0,86084	3,67	0,86075	3,45	0,86101	3,35
983040	3840	256	0,86075	8,70	0,86084	7,38	0,86101	7,05	0,86101	6,96	0,86100	6,90
1966080	3840	512	0,86083	15,11	0,86100	14,57	0,86100	14,30	0,86101	14,27	0,86101	14,22

Tabela 5-14 - Resultados obtidos utilizando 2 GPUs. Meio: ALUMÍNIO

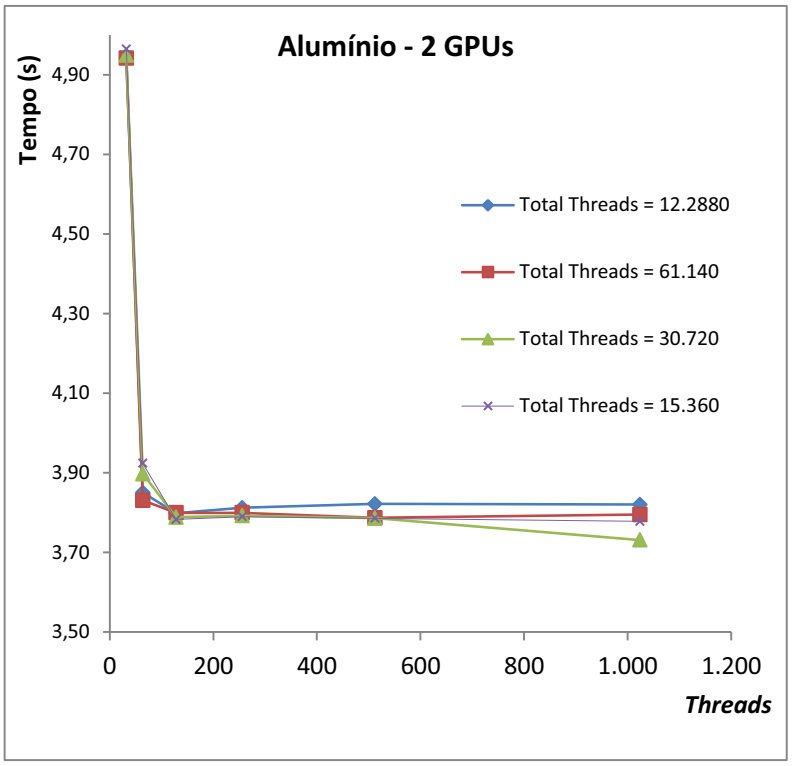

Gráfico 5-8 – Multi-GPUs (2 GPUs). Meio Alumínio

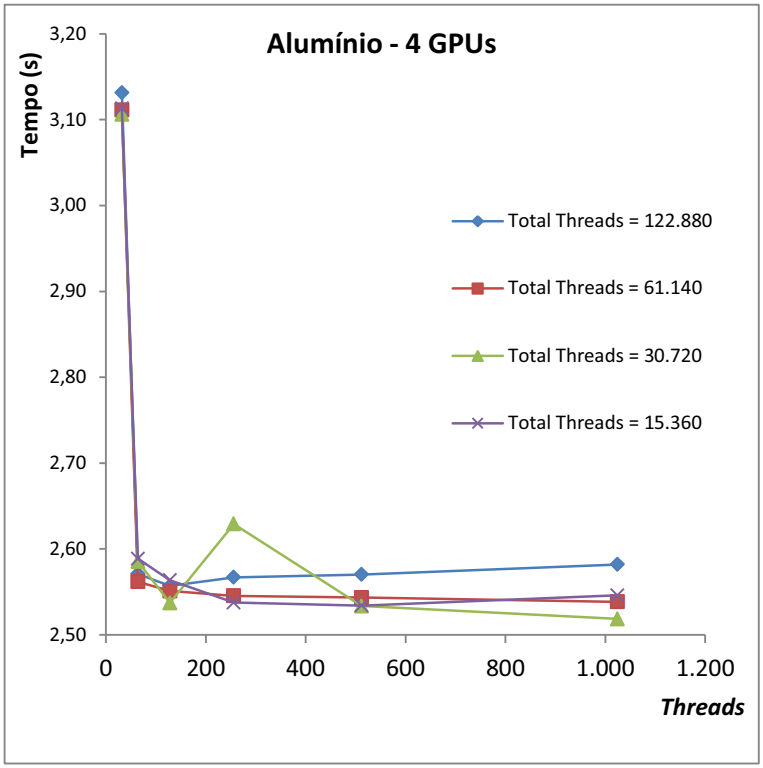

Gráfico 5-9 – Multi-GPUs (4 GPUs). Meio Alumínio

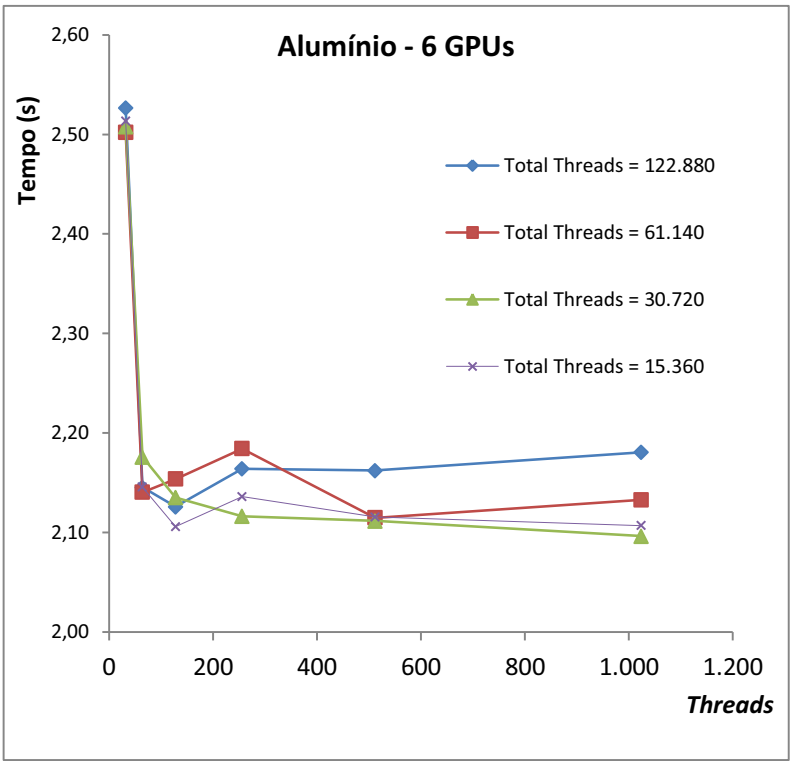

Gráfico 5-10 – Multi-GPUs (6 GPUs). Meio Alumínio

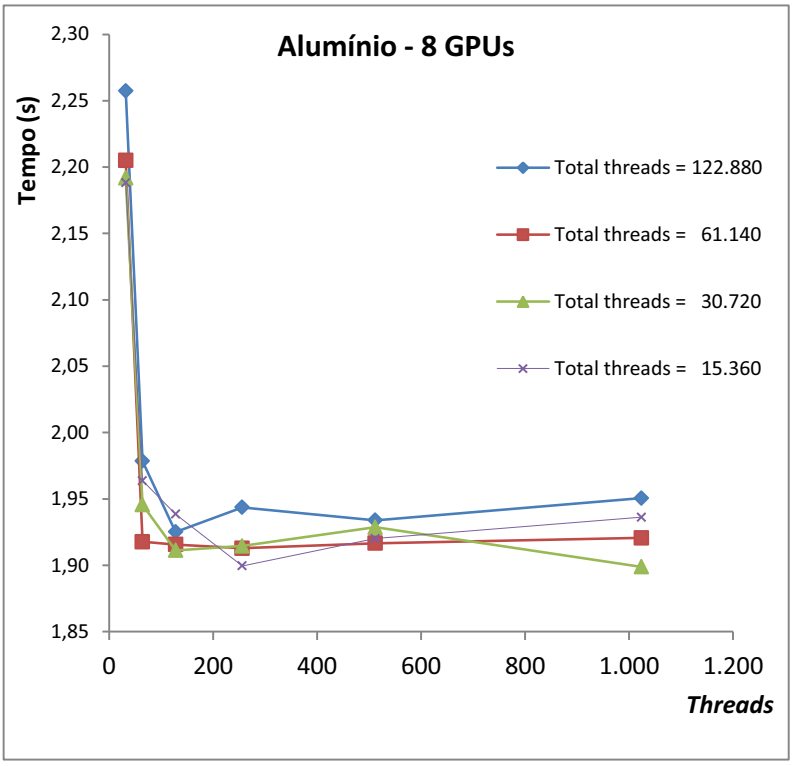

Gráfico 5-11 – Multi-GPUs (8 GPUs). Meio Alumínio

Meio: Cádmio

Fator teórico **0,3198**

Tempo sequencial(seg) **632,42**

Nro total de *threads*	Blocos	*Threads*	1 nó				2 nós		3 nós		4 nós	
			1 GPU		**2 GPU's**		**4 GPU's**		**6 GPU's**		**8 GPU's**	
			Fator	**t(s)**	**Fator**	**t(s)**	**Fator**	**t(s)**	**Fator**	**t(s)**	**Fator**	**t(s)**
15360	480	32	0,31981	3,35	0,31981	2,34	0,31981	1,82	0,31981	1,63	0,31982	1,54
15360	240	64	0,31982	2,75	0,31982	2,04	0,31981	1,65	0,31982	1,55	0,31982	1,49
15360	120	128	0,31982	2,68	0,31981	2,00	0,31982	1,64	0,31982	1,55	0,31982	1,46
15360	60	256	0,31981	2,67	0,31982	2,00	0,31982	1,64	0,31981	1,51	0,31981	1,55
15360	30	512	0,31982	2,67	0,31981	2,00	0,31982	1,64	0,31982	1,52	0,31982	1,45
15360	15	1024	0,31981	2,66	0,31982	1,99	0,31982	1,63	0,31982	1,51	0,31982	1,46
30720	960	32	0,31981	3,35	0,31982	2,33	0,31982	1,81	0,31982	1,63	0,31983	1,57
30720	480	64	0,31982	2,74	0,31982	2,03	0,31982	1,65	0,31982	1,53	0,31982	1,47
30720	240	128	0,31982	2,67	0,31982	1,99	0,31982	1,64	0,31982	1,52	0,31982	1,46
30720	120	256	0,31982	2,68	0,31982	2,00	0,31982	1,65	0,31982	1,51	0,31982	1,48
30720	60	512	0,31982	2,68	0,31982	2,00	0,31982	1,64	0,31982	1,52	0,31982	1,49
30720	30	1024	0,31982	2,68	0,31982	2,00	0,31982	1,64	0,31982	1,54	0,31982	1,46
61440	1920	32	0,31981	3,35	0,31981	2,34	0,31982	1,82	0,31982	1,66	0,31983	1,62
61440	960	64	0,31982	2,68	0,31982	1,98	0,31982	1,63	0,31982	1,56	0,31983	1,48
61440	480	128	0,31982	2,69	0,31982	2,01	0,31983	1,65	0,31982	1,54	0,31983	1,52
61440	240	256	0,31982	2,68	0,31982	2,01	0,31983	1,66	0,31982	1,53	0,31983	1,46
61440	120	512	0,31982	2,68	0,31982	2,02	0,31983	1,65	0,31982	1,52	0,31983	1,49
61440	60	1024	0,31982	2,69	0,31982	2,01	0,31982	1,65	0,31982	1,54	0,31983	1,47
122880	3840	32	0,31982	3,38	0,31982	2,37	0,31983	1,85	0,31984	1,67	0,31983	1,63
122880	1920	64	0,31982	2,72	0,31983	2,03	0,31983	1,67	0,31984	1,55	0,31983	1,50
122880	960	128	0,31982	2,70	0,31982	2,01	0,31983	1,66	0,31983	1,56	0,31983	1,48
122880	480	256	0,31982	2,71	0,31983	2,03	0,31983	1,68	0,31983	1,58	0,31983	1,50
122880	240	512	0,31982	2,70	0,31982	2,03	0,31983	1,66	0,31983	1,56	0,31983	1,51
122880	120	1024	0,31982	2,72	0,31982	2,04	0,31984	1,69	0,31983	1,58	0,31983	1,51

Tabela 5-15 – Resultados obtidos utilizando multiGPUs. Meio CÁDMIO

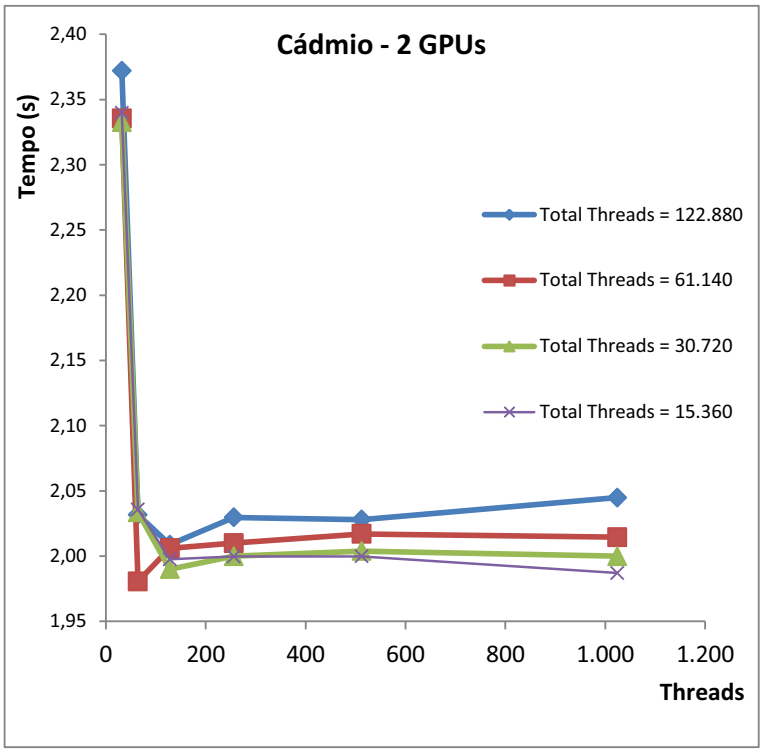

Gráfico 5-12 – Multi-GPUs (2 GPUs). Meio Cádmio

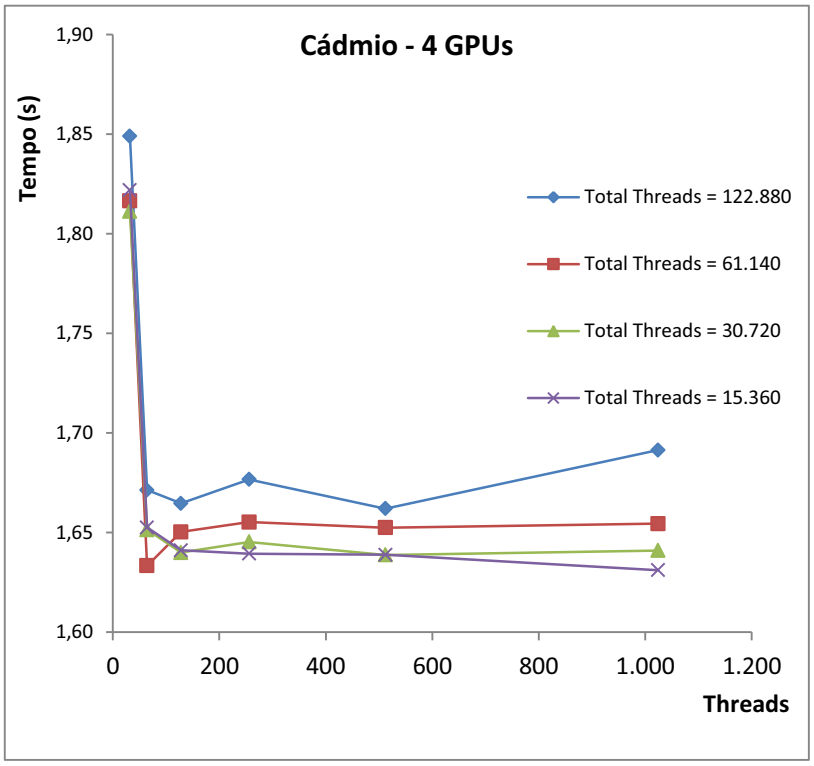

Gráfico 5-13 – Multi-GPUs (4 GPUs). Meio Cádmio

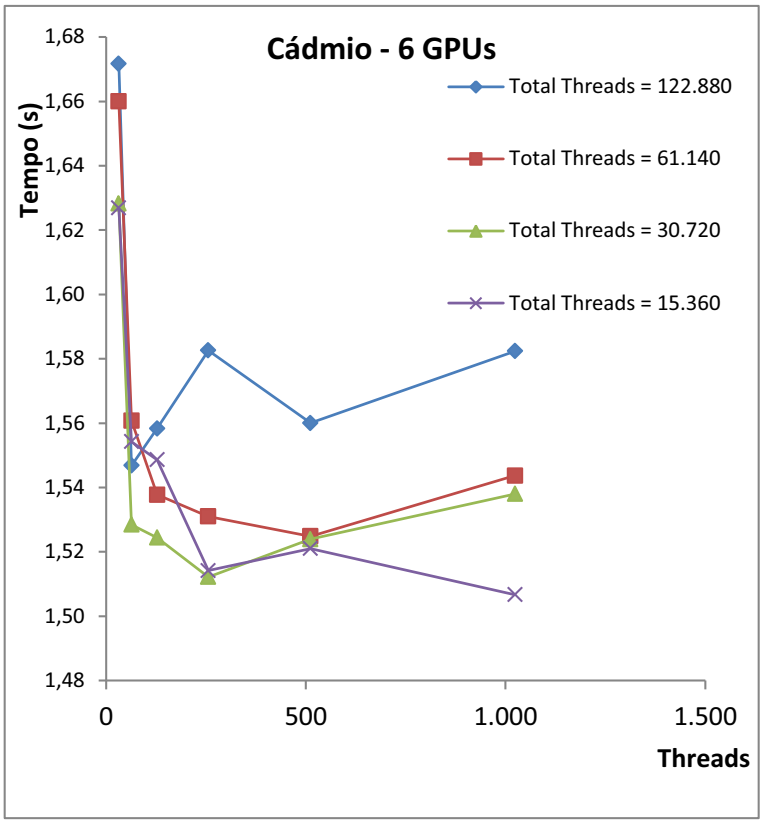

Gráfico 5-14 – MIti-GPUs (6 GPUs). Meio Cádmio

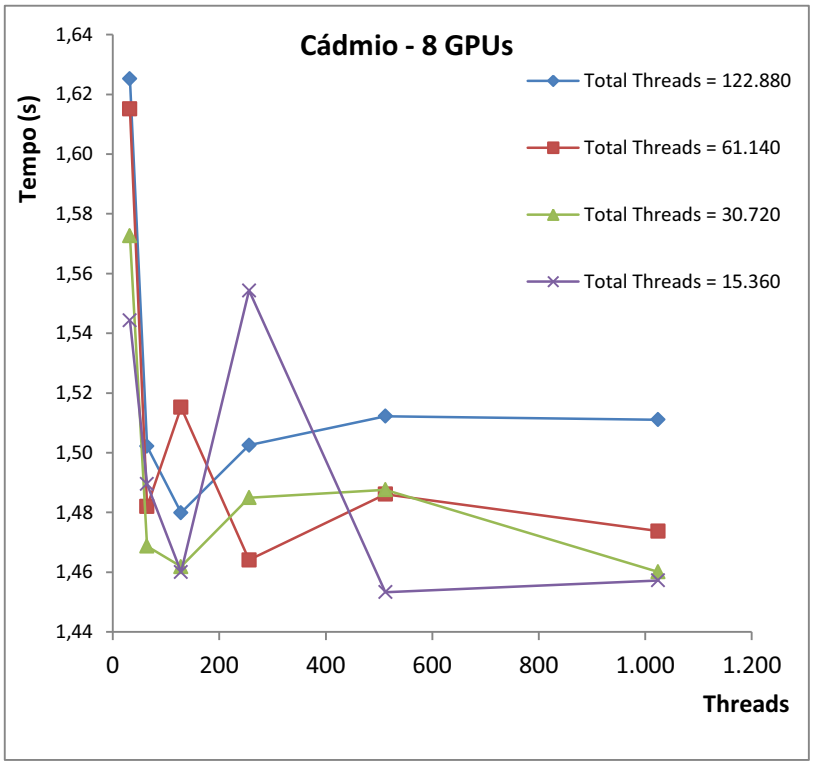

Gráfico 5-15 – Multi-GPUs (8 GPUs). Meio Cádmio

5.4 Influência do MPI no tempo de execução

Observou-se que o tempo encontrado entre os nós pudesse ter uma influencia, em razão da comunicação entre *master* e *slave*. Na Tabela 5-16 vê-se o menor tempo médio da simulação para o transporte de nêutrons usando Monte Carlo para 10^{10} histórias de nêutrons.

Tabela 5-16 – Tempos médios de simulação Monte Carlo por número de nós

Meio	T(s) médio para 1 nó	T(s) médio para 2 nós	T(s) médio para 3 nós	T(s) médio para 4 nós
Água	30,56	15,91	11,05	8,60
Alumínio	3,78	2,53	2,11	1,90
Cádmio	1,99	1,63	1,51	1,45

No Gráfico 5-16 pode-se observar o ganho no desempenho com o aumento do número de nós. Conforme observou-se na Tabela 5-16 o tempo médio para o Meio Água é o que mais exigiu dos recursos computacionais e que mais se beneficiou com o acréscimo do número de nós (cerca de 3,6 vezes mais rápido).

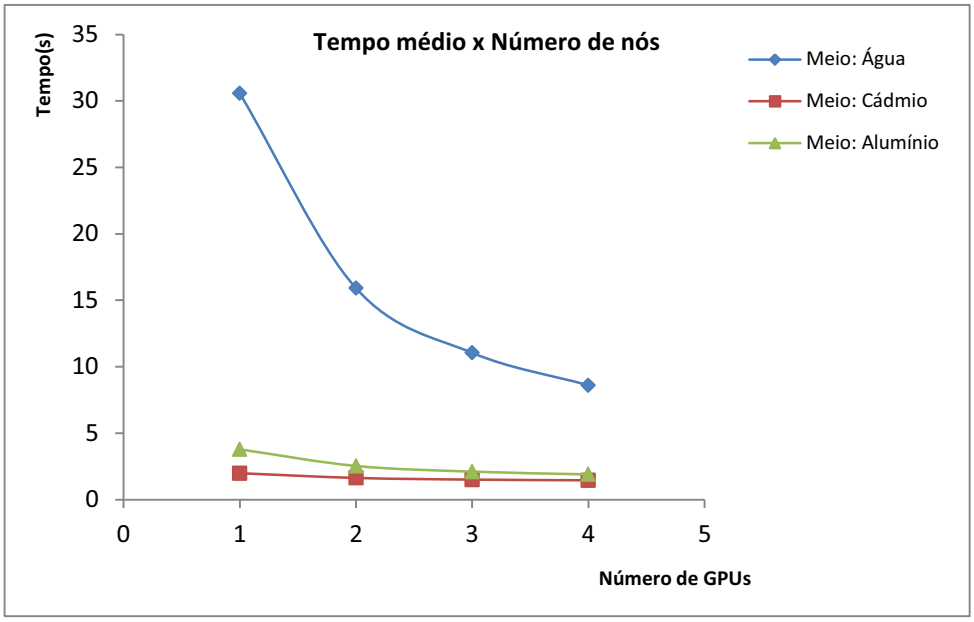

Gráfico 5-16 – Tempo médio em relação ao número de nós para os três Meios

Considerando-se que a diferença encontrada no experimento com 2 GPUs foi de aproximadamente 0,6s e que esta diferença já esteja embutida nos valores encontrados para um nó (2 GPUs), então, podemos encontrar uma diferença de tempo entre o tempo médio experimental e o tempo médio esperado numa situação ideal (sem perdas, em virtude de comunicação), como mostrado na Tabela 5-17.

Tabela 5-17 – Diferenças de tempos em relação a 1 nó

	Acréscimos de tempo(s) entre nós		
Meio	De 1 para 2 nós	De 1 para 3 nós	De 1 para 4 nós
Água	0,63	0,86	3,30
Alumínio	0,64	0,85	0,96
Cádmio	0,64	0,84	0,96

Esses acréscimos de tempo são devido a comunicação entre *master* e *slave* um acréscimo devido ao MPI além do tempo de processamento serial no máster (que introduz um valor fixo). Nos Gráficos 5-17, 5-19 e 5-20 pode-se observar as diferenças entre os tempos encontrados experimentalmente e o tempo desejado.

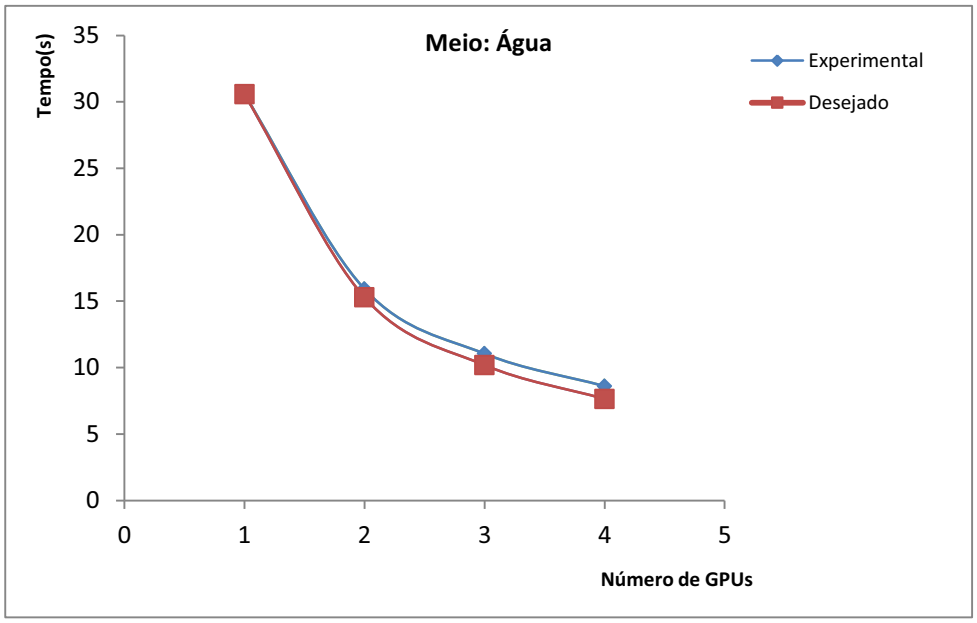

Gráfico 5-17 – Comparativo entre os tempos experimental e desejado. Meio Água

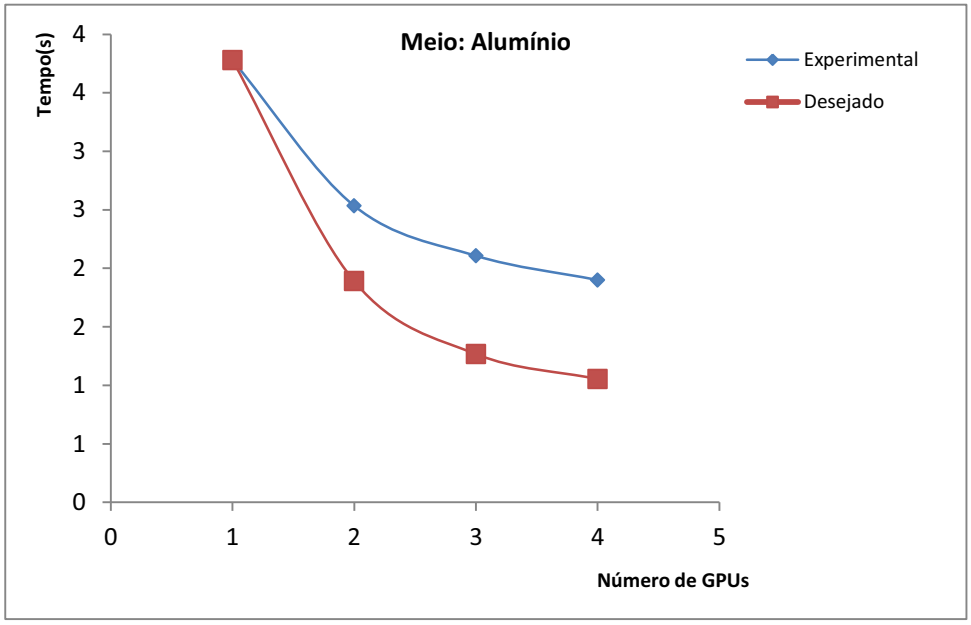

Gráfico 5-18 – Comparativo entre os tempos experimental e desejado. Meio Alumínio

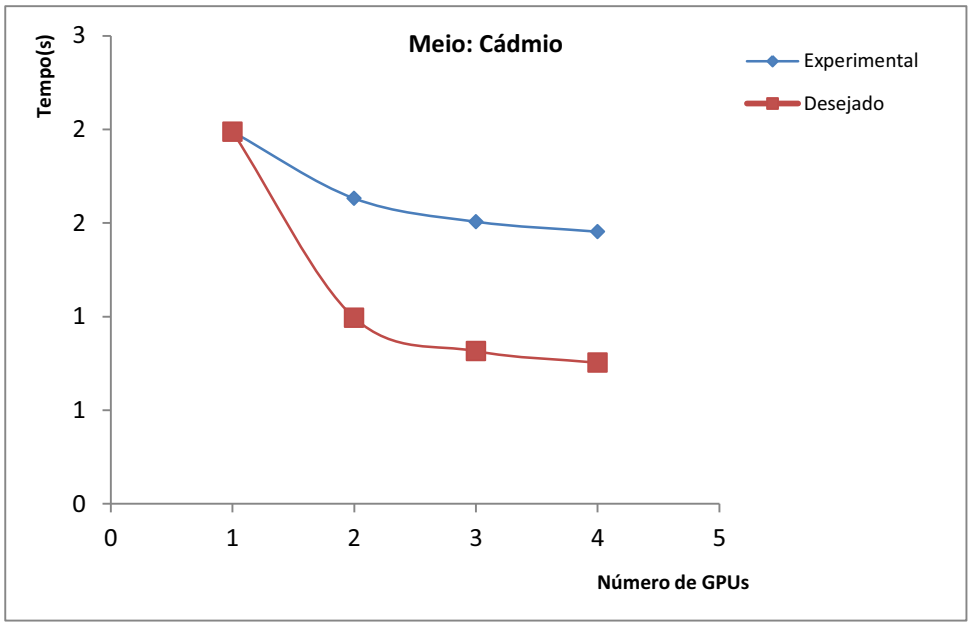

Gráfico 5-19 – Comparativo entre os tempos experimental e desejado. Meio Cádmio

5.5 Comparação entre os tempos encontrados

A Tabela 5-18 expõe a média de três tempos da solução sequencial (CPU), os tempos médios das diversas soluções paralelas para cada Meio e o *speedup* em relação a sequencial.

Tabela 5-18 – Tabela comparativa entre os tempos da solução sequencial e soluções paralela

	Meio: Cádmio		Meio Água		Meio Alumínio	
	Tempo CPU	632,42	Tempo CPU	18.281,57	Tempo CPU	1.749,72
	t (s)	***speedup***	t (s)	***speedup***	t (s)	***speedup***
1 GPU	2,66	237,78	59,80	305,71	6,25	280,17
2 GPUs	1,99	318,28	30,56	598,15	3,78	463,13
4 GPUs	1,63	387,72	15,91	1.149,12	2,53	690,63
6 GPUs	1,51	419,74	11,05	1.655,09	2,11	830,47
8 GPUs	1,45	435,15	8,60	2.124,56	1,90	921,12

Pode-se observar um ganho expressivo em todos os experimentos destacando-se o experimento com o Meio Água com seu *speedup* de 2124. E como esperado os experimentos com menor custo computacional possui menor *speedup* devido aos tempos de comunicação e processamentos sequenciais.

6 CONCLUSÕES E TRABALHOS FUTUROS

A tecnologia GPU, inicialmente destinada a computação gráfica com destaques para os jogos eletrônicos, despertou o interesse da comunidade científica devido à sua grande capacidade computacional de execução de programação em paralelo. Com o advento da CUDA, pela NVIDIA, aumentou ainda mais o interesse por pesquisas em que problemas de alto custo computacional que demandam horas pudessem ser resolvidos em paralelo num tempo muito inferior. Aprender a programar em CUDA demanda conhecer o *hardware* de uma GPU. Dominar conceitos como número de *cores*, multiprocessador *streaming*, *grids,* blocos*, threads*, memória global, *shared* e textura, enfim, fazer uma associação da estrutura de dados com a utilização de *grid*, blocos e *threads* nas dimensões possíveis, é condição imprescindível para se tirar o maior proveito de uma GPU com CUDA.

Por sua vez, reescrever um código sequencial em paralelo nem sempre é possível e fácil; o problema tem de ser paralelizável. O objeto do presente trabalho – problema do transporte de nêutron resolvido por simulação através do método de Monte Carlo – teve uma excelente *performance* em relação à solução sequencial. Tomando-se a água como exemplo, a solução paralela para um número de histórias de nêutrons igual a 10^{10} foi mais de 300 vezes mais rápida para uma GPU e mais de 2.000 vezes para oito GPUs, em relação à solução sequencial. As diferenças encontradas nas soluções de uma para duas GPUs se mantiveram também para múltiplas GPUs.

Por outro lado, por conta do acréscimo de tempo devido à parte sequencial e comunicações (que neste trabalho foi medido e vale 0,6 segundos), um problema de relativo baixo custo computacional poderia não usufruir dos ganhos ou até mesmo ter seu tempo de execução aumentado. Aqui neste trabalho, considerando apenas o acréscimo de tempo para utilização da segunda GPU, o limite seria um problema que levasse 1,2 segundos para execução em uma GPU, pois sua execução em 2GPUs levaria o mesmo tempo, ou seja (1,2/2+0,6) = 1,2 segundos.

Com o modo *persistence mode* ligado este limite poderia ser mais baixo, entretanto, o fato de não funcionar robustamente (erros intermitentes ocorriam) para a GPU utilizada não permitiu sua utilização. Sua utilização será certamente tema de futuras investigações.

Verificou-se outro fato relevante quando se iniciaram os experimentos com um número de histórias e, gradativamente; foi-se aumentando este número, Quando se atingiram 10^9 histórias, os fatores de transmissão encontrados ora estavam muito próximos dos valores teóricos, ora estavam muito distantes. Detectou-se que uma das GPUs estava dedicada a interface gráfica, e resolveu-se, então, executar apenas os experimentos no modo texto. Tudo isto induz à proposta de uma investigação mais profunda em trabalhos futuros.

A busca por melhores desempenhos pelas grandes empresas de tecnologia traz salutares benefícios à comunidade científica, sempre ávida de melhores *performances* nos problemas de simulações que demandam alto custo computacional. A expectativa futura é muito interessante. A Intel investe no projeto Intel Many Integrated Core Architecture (MIC); a AMD propõe uma arquitetura com coprocessador escalar como o futuro das suas GPUs no projeto Fusion; e a NVIDIA, que já vendeu milhões de GPUs, provavelmente deverá acompanhar o mercado. Trabalhos futuros devem acompanhar o impacto da utilização destas novas tecnologias.

Programar em ambiente CUDA-GPU para a área nuclear é algo recente e incipiente. Entretanto, comprovadamente as GPGPUs apresentam grande potencial para ganho de desempenho nos algoritmos. A investigação e desenvolvimento de novas ferramentas que facilitem seu uso de forma mais transparente e amigável, por parte da comunidade científica é possível tema para trabalhos futuros.

7 REFERÊNCIAS

ALMEIDA, Adino Américo Heimlich. **Desenvolvimento de algoritmos paralelos baseados em GPU para solução de problemas na área nuclear**. Dissertação de Mestrado em Engenharia de Reatores. PPGIEN/CNEN, 2009.

BALA, S. KIPNIS, L. Rudolph; SNIR, Marc. Designing e cient, scalable, and portable collective communication libraries. In: **Technical report**. IBM T. J. Watson Research Center, Preprint, Oct. 1992.

_____. Process groups: a mechanism for the coordination of and communication among processes in the Venus collective communication library. In: **Technical report**, IBM T. J. Watson Research Center, Preprint. 2.3, Oct. 1992.

BEGUELIN, A.; DONGARRA, J.; GEIST, A.; MANCHEK, R.; SUNDERAM, V. Visualization and debugging in a heterogeneous environment. In: **IEEE Computer**, 26(6):88-95, Jun. 1993.

BOMANS, Luc; HEMPEL, Rolf. The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2. In: **Parallel Computing**, 15:119-132, 1990.

BORGO, Rita; BRODLIE, Ken. State of the Art Report on GPU Visualization. In: **Visualization & Virtual Reality Research Group**. School of Computing. University of Leeds, 2009.

BUTLER, Ralph; LUSK, Ewing. Monitors, messages, and clusters: The p4 parallel programming system. In: **Parallel Computing**, 20(4):547-564, Apr. 1994. Also Argonne National Laboratory Mathematics and Computer Science Division preprint, p. 362-493.

_____. User's guide to the p4 programming system. In: **Technical Report TM-ANL**. 92/17, Argonne National Laboratory, 1992.

CALKIN, Robin; HEMPEL, Rolf; HOPPE, Hans-Christian; WYPIOR, Peter. Portable programming with the PARMACS message-passing library. In: **Parallel Computing**, 20(4):615-632, Apr. 1994.

CUDA Toolkit 4.0. Curand Guide, Jan. 2011.

DONGARRA, J.; GEIST, A.; MANCHEK, R.; SUNDERAM, V. Integrated PVM framework supports heterogeneous network computing. In: **Computers in Physics**, 7(2):166-175, Apr. 1993.

DUDERSTADT, J. J.; HAMILTON, L. J. **Nuclear Reactor Analysis**. New York: John Wiley & Sons Inc., 1975. p. 606.

EDINBURGH PARALLEL COMPUTING CENTRE. University of Edinburgh. CHIMP Concepts, Jun. 1991.

_____. University of Edinburgh. CHIMP Version 1.0 Interface, May 1992.

FLYNN, M.J. Some computer organizations and their effectiveness. In: **IEEE Transactions on Computers**, v. 24, n. 9, p. 948-960, Sep. 1972.

GEIST, A., BEGUELIN, A.; DONGARRA, J.; JIANG, W.; MANCHECK, R.; SUNDERRAM, V. ***PVM:*** **Parallel Virtual Machine**: A user's guide and tutorial for networked parallel computing. Cambridge, MA: MIT, Press, 1994.

GEIST, G.A.; HEATH, M.T.; PEYTON, B.W.; WORLEY, P.H. PICL: A portable instrumented communications library, C reference manual. In: **Technical Report TM-11130**. Oak Ridge National Laboratory, Oak Ridge, TN, Jul. 1990.

GLASKOWSKY, Peter N. White Paper NVIDIA's Fermi: the first complete GPU Computing Architecture. NVIDIA Corporation, 2009.

GROPP, William D.; SMITH, Barry. Chameleon parallel programming tools user's manual. In: **Technical Report ANL-93/23**. Argonne National Laboratory, March 1993.

HALFHILL, Tom R. White Paper NVIDIA's Next-Generation CUDA Compute and Graphics Architecture, Code-Named Fermi, Adds Muscle for Parallel Processing. NVIDIA Corporation, 2009.

HAMMERSLEY, J.M.; HANDSCOMB, D.C. **Monte Carlo Methods**. Londres: Methuen & Co.; Nova Iorque: John Wiley & Sons, 1964.

HEIMLICH, A.; MOL, A.C.A.; PEREIRA, C.M.N.A. GPU-based Monte Carlo simulation in neutron transport and finite differences heat equation evaluation. In: **Progress in Nuclear Energy** (New series). v. 53, p. 229 - 239, 2011.

HWU, W.-M.W.; KEUTZER, K.; MATTSON, T. The Concurrency Challenge. In: **IEEE Design and Test Computers***,* p. 312-320, jul.-ago. 2008.

KIRK, David B.; HWU, W.-M.W. **Programando para Processadores Paralelos.** Uma Abordagem Prática à Programação de GPU. Tradução de Daniel Vieira. Rio de Janeiro: Campus-Elsevier, 2011.

MOORE, Gorgon E. Cramming more components onto integrated circuits. In: **Electronics Magazine**, v. 38, n. 8, p. 4, April 9, 1965.

MPI: A Message-Passing Interface Standard – Version 2.2. Message Passing Interface Forum. Tennessee: University of Tennessee, Sep. 2009.

NEUMANN, J. von. **First Draft of a Report on the EDVAC***.* Contract No. W-670-ORD-4926, U.S. Army Ordnance Department and University of Pensylvania, 1945; reproduzido em GOLD-STINE, H. H. (Ed.). **The computer:** from Pascal to Von Neumann. Princeton, NJ: Princeton University Press, 1972.

NVIDIA CUDA C Programming Guide, Version 3.2, 22/10/2010.

NVIDIA GeForce GTX 480/470/465 GPUDatasheet, 2010.

OLIVEIRA, R.; CARISSIMI, A.; TOSCANI, S. **Sistemas operacionais**. 3. ed. Série Didática do II-UFRGS, 2004.

PEREIRA, C.M.N.A.; LAPA, C.M.F. Coarse-grained parallel genetic algorithm applied to a nuclear reactor core design optimization problem. In: **Annals of Nuclear Energy**. v. 30, p. 555-565, 2003.

_____. Parallel island genetic algorithm applied to a nuclear power plant auxiliary feedwater system surveillance tests policy optimization. In: **Annals of Nuclear Energy**. v. 30, p. 1.665-1.675, 2003.

PEREIRA, C.M.N.A.; LAPA, Celso Marcelo Franklin; MÓL, Antônio Carlos de Abreu. O impacto do processamento paralelo na otimização de projeto neutrônico de reator através de algoritmo genético. In: **Revista Brasileira de Pesquisa e Desenvolvimento**. v. 4, p. 416-420, 2002.

PEREIRA, C.M.N.A.; SACCO, W. A parallel genetic algorithm with niching technique applied to a nuclear reactor core design optimization problem. In: **Progress in Nuclear Energy** (New series). v. 50, p. 740-746, 2008.

PIERCE, Paul. The NX/2 operating system. In: **Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications**. ACM Press, 1988. p. 384-390.

PRINCÍPIOS BÁSICOS de Segurança e Proteção Radiológica. Universidade Federal do Rio Grande do Sul, set. 2006.

QUINN, Michael J. **Parallel programming in C with MPI and OpenMP**. Oregon: Oregon State University, 2004.

SHREIDER, Yu. A. **Method of Statistical Testing**. Monte Carlo Method, Amsterdam: Elsevier Publishing Company, 1964.

SKJELLUM, A.; LEUNG, A. Zipcode: a portable multicomputer communication library atop the reactive kernel. In: WALKER, D.W.; STOUT, Q.F. (editors). **Proceedings of the fifth distributed memory concurrent computing conference**. Nova Jersey: IEEE Press, 1990. p. 767-776.

SKJELLUM, A.; SMITH, S.; STILL, C.; LEUNG, A.; MORARI, M. The Zipcode message passing system. In: **Technical Report**. Lawrence Livermore National Laboratory, Sep. 1992.

SPAMPINATO, Daniele G.; ELSTER, Anne C.; NATVIG, Thorvald. Modelling Multi-GPU Systems. Trondheim: Norwegian University of Science and Technology (NTNU), s.d.

SUNDERRAM, V.B. PVM, a framework for parallel distributed computing. In: **Concurrency:** Practice and Experience, v. 2, p. 315-339, dez. 1990.

TANENBAUM, A. S. **Organização Estruturada de Computadores**. Rio de Janeiro: Livros Técnicos e Científicos Editora, 2001.

TAUHATA, Luiz *et al.* **Radioproteção e dosimetria**. Rio de Janeiro: CNEN, 2003.

WAINTRAUB, Marcel; SCHIRRU, Roberto; PEREIRA, Cláudio M.N.A. Multiprocessor modeling of parallel Particle Swarm Optimization applied to nuclear engineering problems. In: **Progress in Nuclear Energy** (New series). v. 51, p. 680-688, 2009.

WALKER, D. Standards for message passing in a distributed memory environment. In: **Technical Report TM-12147**. Oak Ridge National Laboratory, Aug. 1992.

WHITE PAPER NVIDIA's Next Generation CUDA Compute Architecture: Fermi. NVIDIA Corporation, 2009.

YORIYAZ, Hélio. **Fundamentos do Método de Monte Carlo para transporte de radiação**. São Paulo: IPEN-CNEN, 2010.