UNIVERSIDADE ESTADUAL DO PIAUÍ
CAMPUS PROFº ALEXANDRE ALVES DE OLIVEIRA
BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO

MARCEL RAIMUNDO DE SOUZA MOURA

CUDA® COMO SOLUÇÃO VIÁVEL EM GPGPU PARA PROCESSAMENTO PARALELO DE ALTO DESEMPENHO

PARNAÍBA – PI
2010

MARCEL RAIMUNDO DE SOUZA MOURA

CUDA® COMO SOLUÇÃO VIÁVEL EM GPGPU PARA PROCESSAMENTO PARALELO DE ALTO DESEMPENHO

Monografia apresentada ao curso de Bacharelado em Ciências da Computação da Universidade Estadual do Piauí – UESPI, Campus Prof. Alexandre Alves de Oliveira, como parte das exigências da disciplina de Estágio Supervisionado, requisito parcial para obtenção do título de Bacharel em Ciências Computação.

Orientador: Eyder Franco Sousa Rios.

Parnaíba – PI
2010

GOVERNO DO ESTADO DO PIAUÍ
UNIVERSIDADE ESTADUAL DO PIAUÍ – UESPI
CAMPUS PROF. ALEXANDRE ALVES DE OLIVEIRA
COORDENAÇÃO DO CURSO DE CIÊNCIA DA COMPUTAÇÃO

Ata de Apresentação de Trabalho de Conclusão de Curso

Aos dez dias do mês de setembro de dois mil e dez, às 08h00, na Sala 201 do Campus Prof. Alexandre Alves Oliveira - UESPI, na presença da Banca Examinadora, presidida pelo Prof. Eyder Franco Sousa Rios e composta pelos membros efetivos os professores Dario Brito Calçada e Daniel Lima Sousa, o aluno **Marcel Raimundo de Souza Moura** apresentou o Trabalho de Conclusão de Curso intitulado **CUDA como Solução Viável em GPGPU para Processamento Paralelo de Alto Desempenho**, como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação. Aberta a sessão pública, o candidato teve a oportunidade de expor o trabalho. Após a exposição, o aluno foi arguido oralmente e avaliado em sessão reservada pelos membros da Banca, nos termos do Regulamento Geral dos Trabalhos de Conclusão de Curso da Universidade Estadual do Piauí, tendo obtido nota 9,3 (nove pontos e três décimos). A Banca concluiu pela aprovação do candidato, **sem restrições**, resultado este divulgado ao aluno e demais presentes. Nada mais havendo a tratar, eu, Prof. Eyder Franco Sousa Rios lavrei a presente ata que será lida e assinada por mim e pelos membros da Banca Examinadora e pelo candidato. Parnaíba(PI), 10 de setembro de 2010.

Banca Examinadora

Prof. MsC Eyder Franco Sousa Rios, UESPI
Presidente

Prof. Esp Dario Brito Calçada, UESPI

Prof. MsC Daniel Lima Sousa, UFPI

Candidato

Marcel Raimundo de Souza Moura

A minha mãe, avós, familiares e amigos por todos os incentivos e apoio ao longo da minha vida.

AGRADECIMENTOS

Agradeço pela força espiritual dada a mim para alcançar metas ao longo da minha vida; a minha mãe Ivete Moura pelo amor, dedicação, educação, apoio e exemplo de conduta; a meus avós e demais familiares por toda educação, afeto, suporte e exemplos de integridade e caráter; a Milton Albuquerque, Maria Helena Albuquerque e seus filhos por todo o apoio, educação e afeto prestados; a Luanne Braúna, minha companheira, amiga e namorada por todo o apoio, companheirismo e amor; aos amigos e companheiros de faculdade, em especial: Conrado Machado e Alessandra Braúna; a todos os professores do curso de Bacharelado em Ciências da Computação da Universidade Estadual do Piauí.

Os problemas significativos que enfrentamos não podem ser resolvidos no mesmo nível de pensamento que estávamos quando os criamos.
(Albert Einstein)

RESUMO

A manufatura de chips processadores está passando por um período de grandes mudanças. Os processadores de um núcleo estão sendo substituídos por arquiteturas multicores, que são tentativas de manter a taxa de crescimento de poder de processamento de dados, sem que haja certeza de eficiência. Na busca por novas formas de processamento paralelo considerar o GPGPU e o seu novo modelo de programação como uma alternativa ao aumento do poder de processamento de dados vem se mostrando viável. Neste trabalho o CUDA® da NVIDIA® é apresentado e analisado como solução para processamento paralelo de alto desempenho em GPGPU. Analisando resultados alcançados por pesquisadores na utilização prática desse sistema heterogêneo em diversas áreas, CUDA® surge como uma opção de baixo custo, fácil implementação sendo capaz de obter ganhos de desempenho expressivos através de alguns princípios simples de otimização, mas sem exigir do programador um conhecimento avançado de programação paralela, tornando transparentes as barreiras impostas pelo processamento paralelo de alto desempenho.

Palavras Chave: CUDA, Processamento Paralelo, Desempenho.

ABSTRACT

The chip manufactoring industry is passing through a period of great changes. The single core processors are being replaced by multicore architectures, that are tentatives to keep the increasing rate of data processing, without ensure of efficience. In the search of new ways to do parallel processing consider GPGPU and your new programming model as alternative to the increase of data processing power is becoming viable. One model that is noticeable in efficiency and applicability is CUDA® from NVIDIA®. In this paper CUDA® from NVIDIA® is presented and analised as a solution for high performance parallel processing in GPGPU. Analising the results achieved by researchers in pratical use of this heterogeneous system in multiple areas, CUDA® is showed as a low cost and easy implementation option and capable of acquire expressive performance gains through simple principles of otimization, but without demand advanced knowledge in parallel programming from programmer, making transparent the barriers imposed by high performance parallel processing models.

Keywords: CUDA, Parallel Processing, Performance.

SUMÁRIO

1 INTRODUÇÃO

A manufatura de chips processadores está passando por um período de grandes mudanças. Os processadores de um núcleo, que a cada geração tinha a densidade de transistores no chip aumentada para otimizar a performance no processamento de dados, não são mais fabricados com o mesmo propósito. Após a comprovação recente, segundo Zhirnov et al. (2003), que a Lei de Moore não pode ser aplicada como prevista na fabricação de chips por conta da dificuldade de dissipação de calor, o aumento das frequências de operação deixou de ser o eixo principal. Desta forma, as indústrias produtoras de processadores apostam nas arquiteturas multicore.

Arquiteturas multicore são tentativas de manter a taxa de crescimento de poder de processamento nos chips, mas segundo Patterson (2010) não há certeza quanto a sua eficiência. A computação está passando por um momento onde a busca por novas formas de processamento paralelo tornou-se vital. Além da necessidade de processamento devemos considerar mais dois fatores importantes nessa corrida: energia e custo. Segundo Kogge et al. (2008), o modelo de processamento paralelo do futuro deve ter alta capacidade de processamento, consumir pouca energia e não ter um custo elevado, ou será abandonada.

Várias soluções estão sendo propostas, dentre elas a utilização de placas de vídeo paralelamente aos processadores convencionais. Utilizar placas gráficas para o processamento de dados genéricos já vinha sendo feito há alguns anos, mas sem muita expressão na área da pesquisa ou desenvolvimento, pois não era fácil utilizar um hardware e sua Interface de Programação de Aplicação (API) para uma função que não era aquela para qual fora desenvolvido.

Inovações nas arquiteturas de hardware de placas de vídeo e o surgimento de um modelo de programação específico para explorar esse hardware vem permitindo o seu uso para processamento de dados paralelamente. Considerar o processamento paralelo com placas de vídeo, conhecido como Processamento Gráfico de Propósito Geral (GPGPU), e novos modelos de programação como alternativa para a manutenção do aumento no poder de processamento de dados pode ser viável. Um modelo que vem mostrando-se eficiente é o Compute Unified Device Architecture (CUDA®) da NVIDIA®. Apesar de suas limitações, a apresentação e análise desse modelo e dos resultados obtidos em sua utilização prática por outros pesquisadores, apresentadas nesse trabalho, podem demonstrar que ele é um modelo viável para mantermos o aumento do poder de processamento em alguns eixos da computação de dados.

Na primeira parte desse trabalho apresentamos um estudo sobre o processamento paralelo de dados desde o início até o surgimento das arquiteturas multicore. Em seguida temos uma análise da arquitetura das GPUs e sua evolução até que o GPGPU fosse possível. No final do segundo capítulo CUDA® é apresentado como um modelo de computação heterogênea de alto de desempenho com GPUs com descrição da arquitetura Fermi de GPUs NVIDIA®, paradigma de programação e Infraestrutura do modelo. O terceiro capítulo é dedicado a uma descrição dos aspectos inerentes ao CUDA®: interfaces de programação, portabilidade, escalabilidade, restrições e otimização. No capítulo seguinte é feita uma análise de resultados obtidos na utilização prática de CUDA® por outros pesquisadores e em seguida considerações sobre a eficiência e aplicabilidade de CUDA® e trabalhos futuros.

2 PROCESSAMENTO PARALELO

Desempenho sempre foi um dos principais objetivos desde que os computadores surgiram. Schaller (1997, apud PARHAMI, 1999) afirma que a Lei de Moore foi originalmente formulada em 1965 e dizia que a complexidade de um chip dobraria a cada ano (posteriormente revisada para 18 meses). Esta lei impulsionou a indústria de manufatura de chips integrados por muitos anos. A Lei de Moore ainda foi revisada recentemente, mas em 2003 foi provado que não seria possível seguir essa tendência de produção por muito tempo, havíamos chegado ao limite dos processadores de um núcleo, vejamos:

> A tendência dos últimos 30 longos anos na microeletrôniça aumentou a velocidade e a densidade através do escalonamento de componentes [...]. Contudo, esta tendência irá terminar à medida que nos aproximamos da barreira de energia devido ao limite quanto a capacidade de dissipação. (ZHIRNOV et al., 2003, p. 1938, tradução nossa).

Na verdade, o limite de transistores por área em um chip não foi atingido ainda, pois a indústria considera que o custo de produção de um chip desse tipo não é vantajoso se comparado com o ganho de performance que será alcançado. Desde que o limite físico da produção de processadores de um núcleo foi previsto, fez-se necessário considerar outros aspectos para manter o ganho de performance no processamento de dados. Arquiteturas de processadores com múltiplos núcleos em frequências menores obtiveram uma relação "performance x custo x consumo de energia" melhor que os seus antecessores de um núcleo, logo a "[...] tecnologia multicore tornou-se o foco no design de CPUs." (PENG, 2007. p.55, tradução nossa).

Arquiteturas com múltiplos núcleos de processamento não surgiram nem foram aperfeiçoadas na última década. Segundo Parhami (1999), em 1966 Flynn-Johnson já propôs uma classificação para sistemas de computação em cadeias de dados ou instruções que continham mais de um núcleo de processamento, mas somente nos últimos dez anos essas soluções passaram a ser implementadas e comercializadas por parte da Intel e AMD. Hoje encontramos processadores de uso doméstico com dois, três, quatro e seis núcleos, e a expectativa de alguns especialistas é que a quantidade de núcleos em um chip dobre a cada dois anos.

Hennessy (2006), perguntado sobre o futuro do paralelismo, afirma que quando falamos de paralelismo e da facilidade de uso do paralelismo real, estamos falando de um problema que é mais difícil que qualquer outro que a ciência da computação já enfrentou. Apostar no paralelismo como o futuro do processamento de dados e ganho de desempenho foi

algo incerto para a ciência da computação. Para Patterson (2006), sobre o mesmo tema, a perspectiva é similar, afirmando que esse foi o maior ocorrido dos últimos 50 anos por que a indústria está apostando o seu futuro como a programação paralela é útil.

O paralelismo nunca foi o alvo dos programadores nem dos fabricantes de processadores até a difusão das arquiteturas multicores e heterogêneas. Isso representa uma mudança total de paradigma dentro da indústria de hardware e software.

Um hardware paralelo só tem o desempenho esperado se for utilizado como tal. Quando as arquiteturas multicore surgiram "com a possível exceção de JAVA®, não haviam linguagens difundidas e usadas comercialmente para desenvolvimento com extensões para multithreading." (CREEGER; 2005 p. 64, tradução nossa), consequentemente não haviam programadores para estas plataformas. Programar para essas novas arquiteturas continua sendo um desafio. Consideremos:

> [...] Programação Paralela é difícil. Nós acabamos com o uso dos comandos GOTO em muitas linguagens [...]. Código paralelo é sujeito a erros como deadlocks e livelocks, condições de corrida, etc. Que podem ser muito subjetivos e difíceis de identificar, também por que o erro é não-repetitivo. [...] Um desafio de igual importância é o escalonamento de performance. A Lei de Amdahl diz que o ganho de velocidade conseguido pelo paralelismo é recíproco à proporção do código que não pode ser paralelizado. (BOYD, 2008, p. 31, tradução nossa).

Várias implementações multicore surgiram nos últimos 10 anos. Todas as empresas de manufatura de processadores voltaram-se para o desenvolvimento de seus próprios chips multicore, são alguns exemplos: Famílias Dual Core, Core 2, Core i da Intel®, Famílias Atlhon X2, Phenom, Opteron, Turion da AMD®, processadores CELL da aliança IBM-Sony-Toshiba, entre outros.

Ao contrário do que naturalmente pensaríamos, tendo em vista a difusão de processadores multicore, "os fabricantes de chips estão ocupados com o design de microprocessadores que a maior parte dos programadores não sabem manipular." (PATTERSON, 2010, p. 24, tradução nossa). A tendência na fabricação de microprocessadores deixou de ser o aumento do número de transistores por área, para focar no aumento de núcleos no mesmo chip.

O ganho de performance hoje é esperado através de paralelismo de múltiplos chips, múltiplos núcleos ou múltiplos contextos. Tratando de paralelismo, Parhamy (1999) afirma que a classificação de Flynn-Johnson define basicamente quatro níveis de paralelismo em hardware. *Single Instrunction Single Data* (SISD), *Single Instruction Multiple Data* (SIMD), *Multiple Instruction Single Data* (MISD), *Multiple Instruction Multiple Data* (MIMD). SISD

representam as máquinas de um núcleo. SIMD são comumente chamados de *Array Processors*, pois as intruções de controle vêm de uma unidade central. Hardwares do tipo MISD são vistos como pipelines genéricos onde a cada estágio é executada uma tarefa relativamente complexa, ao contrário dos pipelines que conhecemos que realizam tarefas simples.

Ainda sobre o mesmo tema, Parhamy (1999) afirma que os hardwares do tipo MIMD são divididos em quatro subcategorias: *Global Memory Message Passing* (GMMP), geralmente não utilizada, *Global Memory Shared Variable* (GMSV), referenciada como multiprocessadores de memória compartilhada, *Distributed Memory Message Passing* (DMMP), também conhecida como multicomputadores de memória distribuída, e, finalmente, a *Distributed Memory Shared Variable* (DMSV) que é um tipo que se tornou bastante popular por conta da fácil implementação. Outros termos utilizados para descrever arquiteturas são relativos à todos os processadores do tipo MIMD executarem o mesmo programa, o resultado é conhecido como *Single-Program Multiple Data* (SPMD) e quando a execução é de vários programas diferentes denominamos *Multiple-Program Multiple Data* (MPMD).

Essas classificações não são temas atuais em processamento de dados, a novidade é utilizar a programação paralela com eficiência, mostrar que ela é uma saída viável para o ganho de desempenho, sem exceder o consumo de energia e custos de produção para manter as necessidades tecnológicas globais.

Kogge et al. (2008) destaca três problemas fundamentais nas microarquiteturas modernas de computadores. Primeiro, a única maneira de conseguirmos um ganho de performance através dos dispositivos que temos atualmente é utilizar paralelismo explícito. Segundo, energia é um recurso a ser considerado. Sistemas que gastam uma quantidade enorme de energia para atingir um ganho de performance mínimo serão abandonados. Terceiro, a diferença de desempenho entre latência e largura de banda de memórias continua crescendo em relação ao ganho de desempenho dos processadores. Arquiteturas heterogêneas que utilizam SPMD e SIMD, tais como GPGPU, oferecem alternativas a esses problemas.

2.1 GPGPU – General Purpose Graphic Processor Unit

As Unidades de Processamento Gráfico (GPUs), informalmente conhecidas como placas de vídeo, são dispositivos desenvolvidos para suprir as necessidades em processamento de mídia - vídeo e imagem. A mídia gerada por uma GPU é formada por figuras geométricas elementares e a menor unidade visual que um dispositivo digital pode gerar é denominada

pixel. As GPUs são utilizadas para processamento de pixels em jogos e outras aplicações que demandam gráficos.

Segundo Kirk e Hwu (2008), a filosofia de design das GPUs é estimulada pelo rápido crescimento da indústria de vídeo games que exerce uma pressão econômica tremenda pela habilidade de realizar uma quantidade massiva de cálculos em jogos avançados. O desenvolvimento de placas de vídeo, devido à pressão da indústria, sempre teve como objetivo criar placas com a capacidade de gerar pixels por unidade de tempo cada vez maior.

A capacidade de processamento pode ser mensurada através da quantidade de operações que um chip é capaz de realizar. Na computação essa medida pode ser feita com operações sobre números inteiros ou pontos-flutuantes. Ponto-flutuante é um formato digital dos números reais, assim como os conhecemos na matemática. A comparação de performance sobre pontos-flutuantes é mais difundida e aceita, pois esse tipo de dado é mais complexo e requer mais recursos computacionais para ser processado. A unidade de medida de operações com pontos-flutuantes é o *Floating-point Operations per Second* (FLOP/s), que representa a quantidade de operações com pontos-flutuantes por segundo. A Figura 1 apresenta uma comparação de desempenho entre Unidades de Processamento Centrais (CPUs) e GPUs em Giga FLOP/s e a Figura 2 uma comparação entre GPUs e CPUs em relação à largura de banda para memória e Gygabytes/s.

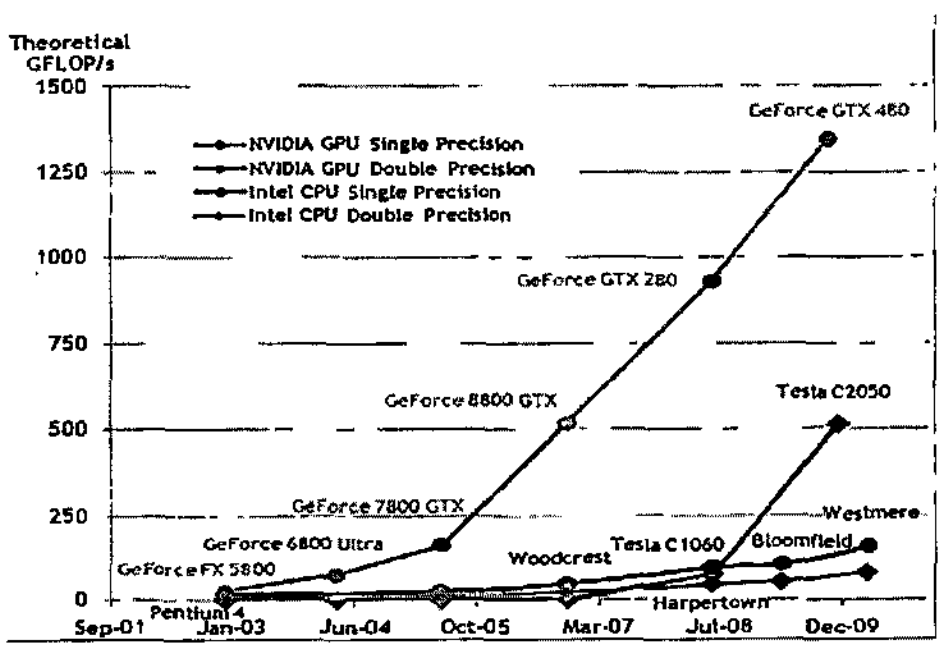


Figura 1: *Gráfico comparativo entre a performance de GPUs e CPUs ao longa dos anos em relação a operações com pontos flutuantes em GFLOP/s. Fonte: NVIDIA CUDA C Progromming Guide, V. 3.1, 2010.*

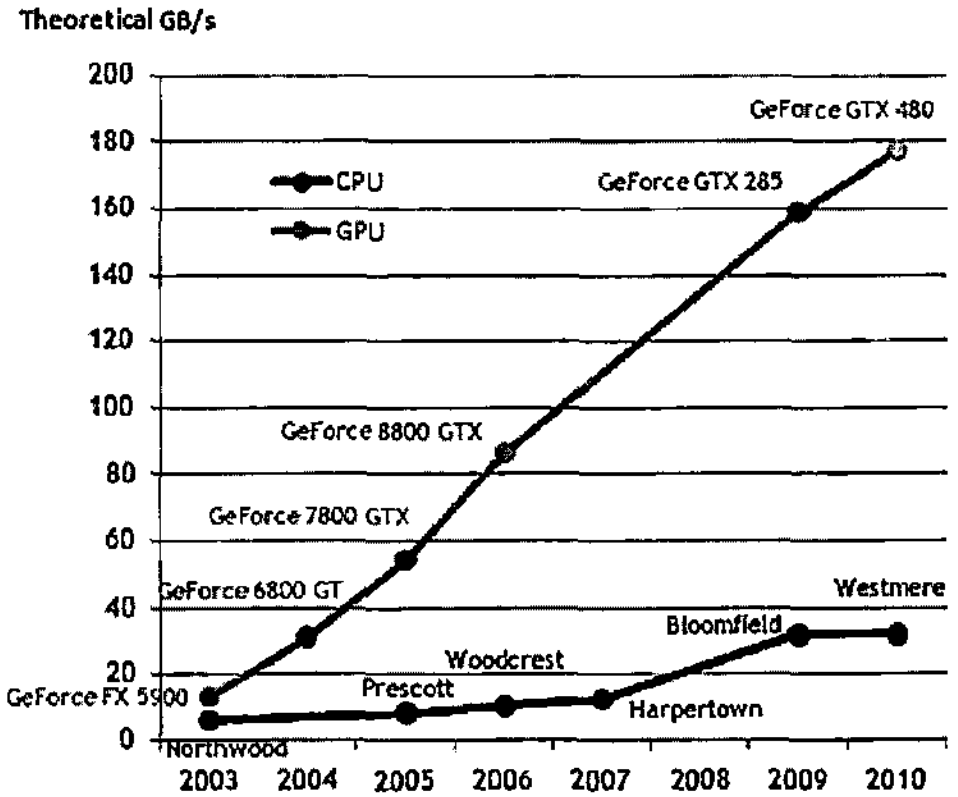


Figura 2: *Gráfico comparativa entre a performance de GPUs e CPUs ao longo dos anos em relação aa crescimento da largura de banda para memória em GB/s. Fonte: NVIDIA CUDA C Programming Guide, V. 3.1, 2010.*

Anaiisando os gráficos fica claro que as GPUs possuem uma capacidade de desempenho muito superior as CPUs, mesmo sendo comparadas com CPUs multicore.

A capacidade de processamento teórica de uma GPU e uma CPU, respectivamente, eram 518 GFLOP/s e 32 GFLOP/s no início de 2008. Ainda em junho do mesmo ano foram introduzidos no mercado chips gráficos que atingiam incríveis 1000 GFLOP/s. Atualmente as GPUs estão com picos de performance em torno de 3000 GFLOP/s.

Essa diferença de performance em favor das GPUs, juntamente com a necessidade de alto poder de processamento, estimulou a idéia de utilizá-las para o processamento de tarefas que seriam destinadas às CPUs, dando início ao surgimento do GPGPU.

GPGPU consiste em utilizar uma GPU paralelamente à CPU, na forma de um sistema heteregêneo, desviando o fluxo de processamento para a GPU quando solicitado pela aplicação e posteriormente retornando os resultados do processamento na GPU e o fluxo de processamento para a CPU. A princípio esse procedimento parece simples, mas foi necessária uma mudança de paradigma de processamento nas GPUs para que esse tipo de sistema heterogêneo fosse possível.

Ao longo dos anos os fabricantes de placas de vídeo desenvolveram vários pipelines

de processamento. Vejamos uma consideração sobre os pipelines gráficos antes da difusão do GPGPU:

> [...] Sistemas gráficos tentam encontrar um equilíbrio apropriado entre objetivos conflitantes de permitir performance máxima e manter uma expressiva, porém simples, interface para descrever computação gráfica. APIs gráficas em tempo real como Direct3D e OpenGL atingem este equilíbrio através da representação da renderização computacional como um pipeline de processamento gráfico que realiza operações em quatro entidades fundamentais: vértices, primitivas, fragmentos e pixels. (FATAHALIAN; HOUSTON, 2008, p. 21, tradução nossa).

Este pipeline de processamento gráfico é divido em sete etapas, das quais três são programáveis. Essas três etapas programáveis apresentam funções específicas que não podem ser feitas por outras etapas e isso gerava uma grande dificuldade para os fabricantes de GPUs. Na figura 3 é apresentado um diagrama dessas etapas.

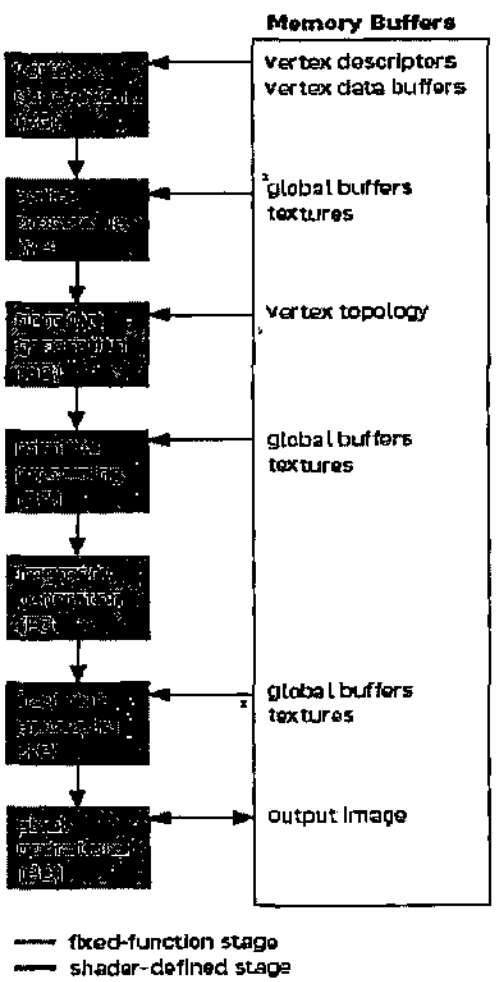


Figura 3: *Gráfico simplificado das etapas de processamento em um pipeline gráfico. Fonte: GPUs a Closer Look, ACM QUEUE, Vol. 6, No. 2, p. 19-28, 2008.*

A utilização menor ou maior de uma etapa do pipeline dependia de algumas características inerentes aos pixels que estavam sendo gerados. Assim os fabricantes tinham que buscar um equilíbrio na divisão de recursos de processamento da GPU entre as etapas do pipeline, mas isso era completamente imprevisível, já que não era possível prever que tipo de aplicação utilizaria a GPU e muito menos as características dos pixels gerados por ela. "Contudo, recentemente os fabricantes decidiram que era um desperdício de transistores restringir os pipelines a um único tipo de processamento [...]" (BYDAL, 2008, p. 3, tradução nossa).

Além da dificuldade no equilíbrio dos recursos da GPU, o custo e esforço de desenvolvimento envolvidos eram altos, além do hardware não ser flexível – não se adaptava facilmente a evolução de novos algoritmos e programas de processamento de mídia. Com o objetivo de contonar todas as barreiras surge a motivação para o uso de processadores de fluxo programáveis, que são núcleos de processamento em série de utilização genérica.

O uso de processadores de fluxo solucionou os dois principais problemas dos fabricantes. Não havia a necessidade de dividir os recursos da placa entre os processadores de fluxo, pois todos eram capazes de realizar qualquer função programável na GPU e compartilhavam todos os recursos disponíveis. Agora o hardware é completamente flexível e programável. Essa mudança de paradigma arquitetural nas placas de vídeo e a necessidade de melhorar o desempenho das CPUs contribuiu para o avanço de pesquisas e estudos com GPGPU.

As primeiras GPUs não tinham hardware programável, mesmo assim alguns pesquisadores conseguiam fazer cálculos não-gráficos. As primeiras utilizações das GPUs para processamento genérico aconteceram antes da aplicação do conceito de processadores de fluxo no pipeline gráfico. Ainda utilizando processadores não programáveis – funções fixas – surgiram, segundo Owens (2007), aplicações como:

- planejamento de movimentação de robôs (1990),
- cálculo de diagramas de Voronoi (1997),
- detector de proximidade (2001),
- redes neurais (1998)
- convoluções e transformadas Wavelet (1999).

A primeira geração de GPUs programáveis não era tão poderosa com as conhecemos atualmente. Em sua primeira geração não existia a possibilidade de processar pontos-flutuantes, mas os trabalhos continuavam a serem produzidos. Ainda, segundo Owens (2007), podemos destacar os trabalhos de multiplicação de matrizes com placas de vídeos NVIDIA®

(2002), solução da equação de Navier-Stokes (2004) e solução de equações diferenciais parciais (2004).

Os processadores de fluxo capazes de processamento de pontos flutuantes transformaram as GPUs em dispositivos altamente paralelos e versáteis, capazes de gerar centenas de GFLOPs de dados processados.

Trabalhar com as linguagens que foram criadas para gráficos de uma forma que não era o foco delas, era uma atividade complicada. Os pesquisadores tinham que fazer uma placa de vídeo (criada para processamento de mídia) ser utilizada para aplicações genéricas através de linguagens que não foram desenvolvidas com tal propósito. Não podemos deixar de lembrar que as mesmas dificuldades de programação paralela que existem para arquiteturas multicore persistem no GPGPU.

A tarefa torna-se mais desafiadora por se tratar de um sistema heterogêneo para o paralelismo, pois a GPU é um núcleo de processamento simétrico multicore que é exclusivamente acessado e controlado pela CPU. A GPU processa fora de sincronia em relação à CPU, possiblitando execução paralela e transferência de dados através da memória principal do sistema.

A dificuldade da programação paralela para sistemas multicore fez considerar-se um outro paradigma de processamento paralelo, o paralelismo de dados. Vejamos o que Boyd Chass destaca sobre esse assunto:

> [...] a hora de vislumbrarmos um novo paradigma chegou – idealmente um que seja escalável com a contagem de cores, mas sem requerer uma reestruturação da arquitetura da aplicação toda vez que uma nova quantidade de cores seja atingida. Assim sendo, isto não é sobre escolher um paradigma que funcione bem com uma quantidade fixa de cores; é sobre escolher um que continue a escalonar com o aumento do número de cores sem precisar de mudanças no código. (BOYD, 2008, p. 32, tradução nossa).

No paralelismo de dados, ao contrário do que acontece no paralelismo de tarefas – que tenta dividir um sistema em subtarefas que serão executadas por vários núcleos – a divisão acontece em nível de dados. Os dados são divididos entre os núcleos e processados paralelamente. Os sistemas de paralelismo de dados que eram desenvolvidos anteriormente voltaram a ter visibilidade com o suporte das GPUs.

Atualmente as GPUs da NVIDIA® são sistemas multicore utilizados para processamento paralelo de dados genéricos. O desenvolvimento de uma plataforma e placas de vídeo que permitissem o seu uso para o processamento paralelo e genérico de dados de uma forma simples foi o foco da NVIDIA® ao desenvolver CUDA® e a arquitetura TESLA®.

2.2 CUDA – Compute Device Unified Architecture

As primeiras GPUs desenvolvidas pela NVIDIA®, que haviam sido utilizadas para GPGPU, não apresentavam propriedades que permitissem o seu uso para computação genérica de uma maneira simples. À medida que a computação genérica com placas de vídeo tornava-se algo promissor os problemas desse tipo de arquitetura surgiam.

Podemos destacar três problemas principais. O primeiro, o programador deveria ter um conhecimento profundo da API e arquitetura da placa de vídeo. Era necessário que o programador soubesse como funcionavam e como estavam organizados os cores e as memórias, para conseguir algum ganho de performance. Segundo, a complexidade de um programa genérico aumentava drasticamente quando era expressado na forma de vertex, texturas ou shaders. Transcrever o código de um programa genérico utilizando estruturas de dados diferentes e fazer com que seja processado por APIs gráficas é uma tarefa complicada e requer repetitivos testes para atingir algum resultado. Terceiro, algumas funções básicas da programação convencional não eram suportadas, tais como leitura e escrita aleatória na memória do dispositivo. A falta de possibilidade em implementar alguns recursos básicos da computação genérica tornava alguns procedimentos impossíveis de serem executados em uma GPU.

A NVIDIA® lançou CUDA® em 2006 como uma arquitetura de computação software-hardware baseada em uma extensão da linguagem de programação C, que permite acesso às instruções e controle da memória da GPU paralelamente à CPU. Até o lançamento de CUDA®, GPGPU não tinha prestígio entre os pesquisadores e programadores. As diferenças de arquitetura e paradigmas de programação entre CPU e GPU existentes até a adoção dos processadores de fluxo em placas de vídeo, tornavam os programadores que desenvolviam aplicações com vídeo muito especializados naquele tipo de desenvolvimento, enquanto a maioria dos programadores de aplicações genéricas não tinha a menor idéia de como utilizar o enorme potencial de processamento paralelo de uma GPU.

CUDA® permite aos programadores utilizar o potencial das placas de vídeo de uma maneira simples, que possibilita portabilidade, flexibilidade, escalonamento, abstração e integração com as aplicações já existentes. Vejamos o que Nickolls afirma:

> De acordo com o conhecimento convencional, programação paralela é difícil. Experiências prévias com o modelo de programação paralela escalável CUDA e a linguagem C, contudo, mostram que os programas mais

> sofisticados podem ser rapidamente expressados com um simples e fácil entendimento da abstração. (NICKOLLS et al., 2008, p. 41, tradução nossa).

A arquitetura unificada permite desenvolver algoritmos que podem ser executados em GPUs NVIDIA® a partir da série GeForce® 8, Tesla® e Quadro® até as mais atuais, apenas com um conhecimento do modelo de programação e sem a necessidade de uma linguagem de programação completamente nova.

O nível de abstração alcançado com CUDA® torna a programação paralela com GPUs completamente possível para qualquer programador. Três elementos principais de abstração constituem o núcleo do modelo: hierarquia dos grupos de threads[1], memória compartilhada e barreiras de sincronização. Esses três aspectos são controlados pelo programador de uma maneira direta utilizando principalmente extensões da linguagem de programação C (nas versões mais recentes de CUDA® existe suporte a outras linguagens de programação) através de um conjunto limitado de instruções.

2.2.1 Arquitetura Fermi

As arquiteturas de GPUs NVIDIA® evoluíram significativamente desde a sua primeira versão que dava suporte a CUDA®. A primeira versão surgiu nas placas da série G80, que dentre outras características, podemos destacar como sendo as primeiras placas de vídeo com suporte à linguagem de programação C e a integrar os pipelines de vertex e pixels em um só (aplicação do conceito de processadores de fluxo), também introduziu memória compartilhada com o host e uma barreira de sincronização para processamento paralelo, facilitando a coordenação de threads.

A geração GT200 é a segunda geração de placas NVIDIA® com suporte a GPGPU. Trata-se de uma revisão da arquitetura anterior com a quantidade de processadores de fluxo aumentada de 128 para 240, – nessa arquitetura os processadores de fluxo passaram a ser referenciados como cuda cores – a quantidade de memória de alto desempenho vinculada a cada processador teve seu tamanho aumentado em relação à arquitetura G80, uma nova política de acesso à memória foi implementada, aumentando a eficiência e o suporte a pontos flutuantes de precisão dupla, o que permitiu o uso de CUDA® em aplicações científicas e de alto-desempenho que antes não podiam ser implementadas.

---

[1] Thread é uma porção menor de um código em execução que foi gerado para execução paralela.

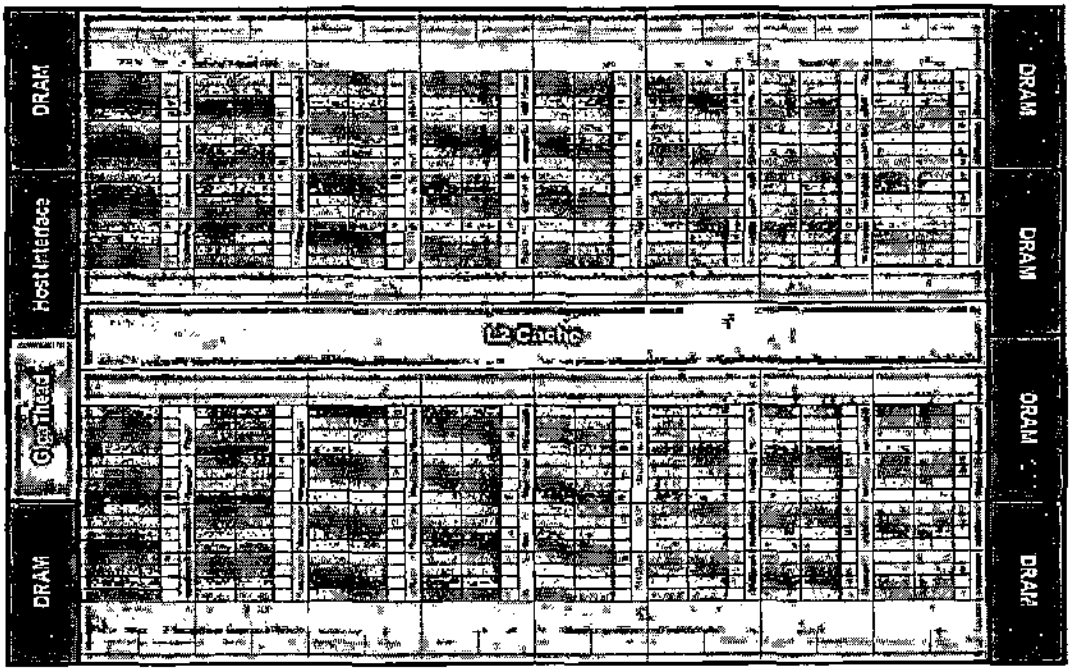


Figura 5: *Arquitetura de GPUs NVIDIA® com 16 SMs posicionados ao redor de um cache L2 comum, memória global DRAM, Interface com o host e o cantralador global GigaThread. Cada retângulo vertical corresponde a um SM com seus controladores próprios (em laranja), unidades de execução (em verde) e um cache L1 com registro (em azul claro). Fonte: NVIDIA's Next Generation CUDA Compute Architecture: Fermi, V. 1.1, 2009.*

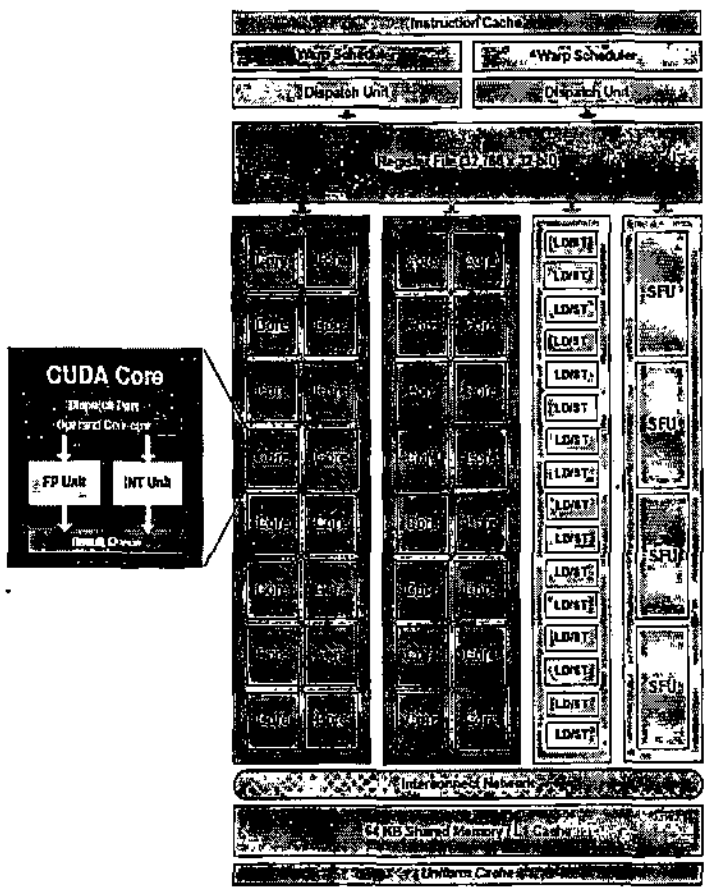
A arquitetura interna dos SMs sofreu modificações, o que não havia acontecido nas duas gerações anteriores:

- Cada núcleo CUDA® na arquitetura Fermi® possui uma Unidade Lógica Aritmética (ALU) e uma Unidade de Pontos Flutuantes (FPU) completamente independentes;
- Implementa o padrão IEEE 754-2008 para operações com pontos flutuantes. Esse padrão define a maneira como as operações com pontos flutuantes devem ser executadas em CPUs e Unidades de Processamento de Pontos Flutuantes (FPUs) e implementações de software. Uma de suas características principais é a redução da perda de precisão em operações com pontos flutuantes;
- Capacidade de processar números com 32 bits de precisão para todas as instruções, como a maioria das linguagens de programação requer.
- Conta com 16 unidades de Load/Store, ao contrário das outras versões onde uma única era compartilhada por todos os SMs. Cada unidade permite o cálculo de endereços de destino e chegada de dados, utilizando 16 threads por clock e ainda suportam leitura e escrita simultânea em endereços da memória cache ou DRAM compartilhada;
- Cada SM com quatro unidades de execução de funções especiais (SFUs) que executam funções como seno, coseno, raiz quadrada etc.

- Na arquitetura Fermi também acontece a introdução de um novo controlador de warps[2]. Cada SM conta com dois controladores de warps e duas unidades de despache de instruções, permitindo que cada SM execute até dois warps concorrentemente e completamente independentes.

A maior inovação da arquitetura Fermi aconteceu em relação ao cache dos SMs. Esse cache foi aumentado para 64KB e ainda pode ser configurado de duas maneiras: 16KB de cache L1 e 48KB de memória compartilhada ou 48KB de cache L1 e 16KB de memória compartilhada. Essa das memórias de acesso rápido é determinante para o melhoramento da arquitetura. Na série GT200 o cache era de apenas 16KB o que forçava os programadores a acessarem a memória global da placa, levando-a a uma queda de desempenho. A possibilidade de controlar a quantidade de memória no cache L1 e compartilhado também permite um nível de customização da aplicação, mas em compensação leva a considerações mais complexas quanto ao gerenciamento de memória.

A Figura 6 oferece uma visão completa com todos os elementos novos introduzidos/modificados na arquitetura Fermi®.



[2] Grupos compostos de 32 threads que são executadas paralelamente no mesmo SM.

Figura 6: *Elementos internos de uma GPU NVIDIA® arquitetura Fermi®. Fonte: NVIDIA's Next Generation CUDA® Compute Architecture: Fermi®, V. 1.1, 2009.*

2.2.2 Paradigma de CUDA

CUDA® proporciona ao programador um nível de abstração da API de vídeo de modo que programar no modelo unificado torna-se semelhante à programação serial. CUDA® pode ser utilizada atualmente com várias linguagens de programação, mas seu modo de utilização mais comum é como uma extensão das linguagens convencionais C e C++.

O programador é livre para escrever o código serial e a qualquer momento declarar uma função, chamada kernel, que será executada paralelamente na GPU. kernels são semelhantes às funções convencionais e podem executar códigos simples ou programas inteiros. A principal diferença do kernel para uma função convencional em C, é que ele necessita do modificador *__global__* em seu cabeçalho e de uma declaração explícita da quantidade de threads que serão usadas para processá-lo, através do modificador <<<...>>>, no momento da chamada ao kernel.

Na amostra de código da Figura 7 dois vetores de tamanho N, A e B, são somados e o resultado é gravado no vetor C, *VecAdd* é um kernel que será executado paralelamente na GPU. Dentro do kernel é utilizada a variável local *threadIdx* que identifica as threads criadas. Observemos ainda na chamada ao kernel, no bloco principal da aplicação, a especificação das threads que serão utilizadas para a execução do kernel.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```

Figura 7: *Amostra de código de um kernel. Fonte: NVIDIA® CUDA® C Programming Guide, V. 3.1, 2010.*

As threads são organizadas em blocos referenciados como *thread blocks* (blocos de thread). Cada bloco de thread, na arquitetura Fermi®, pode conter até 1024 threads, onde cada uma delas possui um identificador único chamado de thread id. Utilizando este identificador

através da variável local ao kernel *threadIdx*, que é um vetor de três componentes, onde a variável precisa ser indexada para identificar uma thread específica, o programador pode controlar todas as threads de um bloco. Contudo, o limite de threads em um bloco não é o limite de threads por kernel. Um kernel pode ser executado por um conjunto de blocos de thread, e esses blocos terão a mesma quantidade de threads, essa é uma característica do modelo de programação.

Os blocos de thread são organizados em grids de uma ou duas dimensões, como mostrado na Figura 8. O número de blocos de thread em um grid geralmente é determinado pelo tamanho dos dados que estão sendo processados ou quantidade de processadores no sistema. Uma aplicação pode executar mais de um grid independentes ou não, sendo limitada a quantidade de recursos de hardware disponíveis. Grids independentes podem ser executados em paralelo e os dependentes em série utilizando uma barreira de sincronização entre eles, ou seja, o próximo grid só começa a ser executado após o primeiro terminar de ser processado. Essa política de sincronização evita problemas comuns no paralelismo de sistemas heterogêneos.

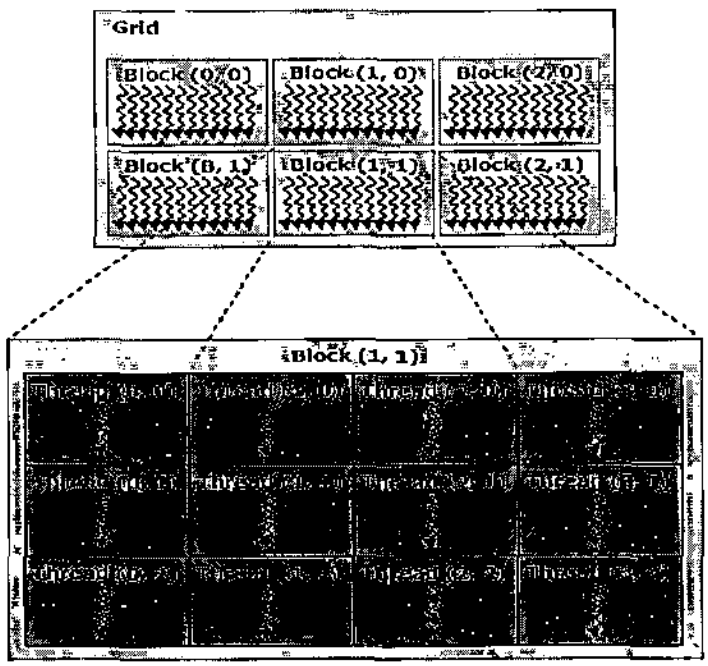


Figura 8: *Grid de blocos de thread organizados e indexados em duas dimensões. Fonte: NVIDIA® CUDA® C Programming Guide, V. 3.1, 2010.*

Blocos de thread podem ser executados independentemente em qualquer ordem, em paralelo ou em série. As threads que compõem um bloco podem cooperar umas com as outras trocando dados através da memória compartilhada e coordenando o acesso a memória. O programador pode definir em um kernel alguns pontos de sincronização que são utilizados

para garantir que a partir daquele ponto de execução, todas as funções anteriores tenham sido executadas por todas as threads de um bloco. Esse conceito é chamado de barreira de sincronização e é implementado através da chamada à função *__syncthreads()*.

Observe que a quantidade de threads de um bloco pode ultrapassar a quantidade de threads por warp ou mesmo a quantidade de cores de um SM, porém, um SM pode trocar de warp sem causar nenhuma espera no sistema, podendo executar todas as warps de um bloco ou mesmo de blocos diferentes. A quantidade de warps diferentes que um SM pode executar é limitada somente pela memória utilizada no processamento.

É importante perceber que toda a execução paralela, agendamento e terminações de processamentos são controlados de forma transparente ao programador e todo o controle das threads é feito diretamente no dispositivo, sem necessitar de nenhum processamento por parte do host.

Durante a execução de um kernel na GPU, uma thread pode acessar dados de vários locais diferentes. Até a arquitetura G200, cada thread possuía uma memória privada local, bem como podia acessar a memória compartilhada por todas as threads que formam o bloco, além do acesso a memória global, também compartilhada. A Figura 9 demonstra essa estrutura de hierarquia de memória. O espaço de memória reservado para um thread e para um bloco de threads só existe enquanto estas entidades existirem. Para designar se uma variável deve ser alocada na memória compartilhada ou na memória global faz-se uso dos modificadores *__shared__* e *__device__*, respectivamente. Essas memórias são distintas fisicamente, as memórias compartilhadas são memórias de baixa latência próximas aos SMs, enquanto a memória global refere-se a uma memória de acesso mais lento que corresponde à memória DRAM da placa.

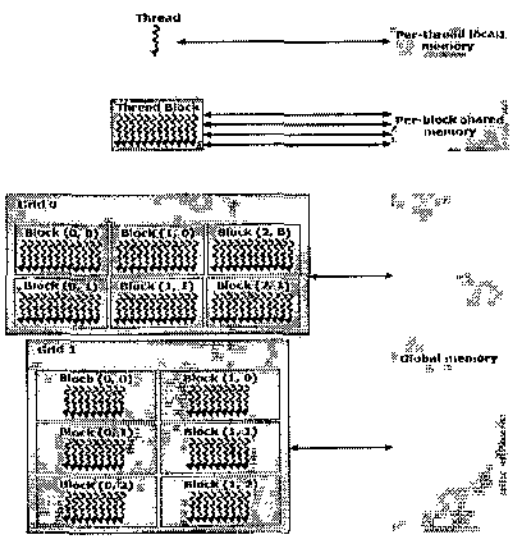


Figura 9: *Hierarquia de memória em uma GPU GT200 com suporte a CUDA®. Fonte: NVIDIA® CUDA® C Programming Guide, V. 3.1, 2010.*

Na arquitetura Fermi® a mudança acontece com a introdução de mais dois caches de memória. Algumas aplicações funcionam bem com o modelo anterior, onde só havia a memória compartilhada por blocos, sem a existência de um cache individual para o SM. Na arquitetura atual, como apresentado na Figura 10, é implementado um modelo de caminho unificado de solicitação para leitura e escrita através de um cache L1, privado ao SM, e um cache L2, compartilhado através dos SMs, e que serve a todas as operações de leitura e escrita. A quantidade de memória para o cache L1 e para a memória compartilhada é configurada pelo programador, permitindo um alto grau de customização para a aplicação a fim de alcançar o melhor desempenho possível.

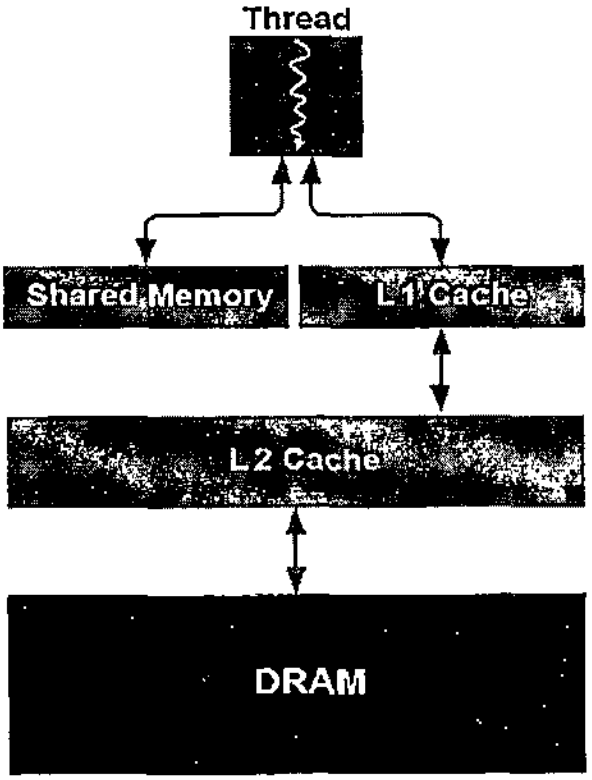


Figura 10: *Madelo de hierarquia de memória na arquitetura Fermi®. Fonte: NVIDIA® CUDA® C Programming Guide, V. 3.1, 2010.*

O gerenciamento de memória de uma aplicação CUDA® é feito através de chamadas explícitas, utilizando comandos para alocação, *cudaMalloc()*, liberação *cudaFree()* e como quase sempre é necessária a transferência de dados para a GPU, é utilizado o comando *cudaMemcpy()*, que permite a cópia de dados entre o espaço alocado e o host.

CUDA® é um sistema similar ao *Single Program Multiple Data* (SPMD), onde o paralelismo é expressado de maneira explícita, controlado pelo programador, e cada kernel executa uma quantidade de threads limitadas. Isso dá ao programador a possibilidade de utilizar a quantidade de threads que for conveniente para aquela porção do código, ao contrário de outros sistemas SPMD onde o código deve adequar-se ao modelo de processamento.

CUDA® dá ao programador a oportunidade de vários níveis de paralelismo, como destaca Nickolls:

> Threads concorrentes de um mesmo bloco de threads representam o paralelismo de fina granularidade de dados e threads. Os blocos de threads independentes de um mesmo grid representam o paralelismo de dados de granularidade grossa. Grids independentes representam o paralelismo de tarefas de granularidade grossa. Um kernel é um simples código em C para uma thread dessa hierarquia. (NICKOLLS, 2008, p. 47, tradução nossa).

Todos esses tipos e níveis de paralelismo são definidos pelo programador ao definir um Kernel, tornado CUDA® flexível e facilmente adaptável a um código serial.

2.2.3 Infraestrutura

CUDA® é uma arquitetura heterogênea de processamento paralelo do tipo SIMT (Single Instruction Multiple Threads), similar à arquitetura SIMD. Na execução de um código em um sistema heterogêneo, existe outro núcleo de processamento como parte do sistema que auxilia a CPU. O processamento tem início na CPU, através de um programa serial em uma linguagem convencional de programação. Nesse código serial estarão inseridos elementos que farão com que o fluxo de processamento de dados seja desviado para o outro elemento da arquitetura, ou esse controle pode ser feito de forma transparente, dependendo da arquitetura.

No caso específico de utilização de GPGPU, nem todas as GPUs são capazes de processar paralelamente à CPU. Em CUDA®, para que uma placa seja utilizada como tal é necessário que ela tenha suporte a essa tecnologia. Quase todas as GPUs lançadas desde a série G80 da NVIDIA® oferecem esse requisito, com exceção das placas de baixo custo. O custo de uma GPU NVIDIA® com suporte a CUDA®

Além da placa é necessário um host com uma CPU compatível capaz de integrar a placa em seu sistema através de um slot de expansão PCI Express 2.0, permitindo taxas de transmissão de dados na ordem de 16GB/s entre o host e sua memória até a GPU. Até o

terceiro trimestre de 2010 deve ser lançado no mercado o novo padrão PCI Express 3.0 que dobrará a taxa de transferência da versão atual.

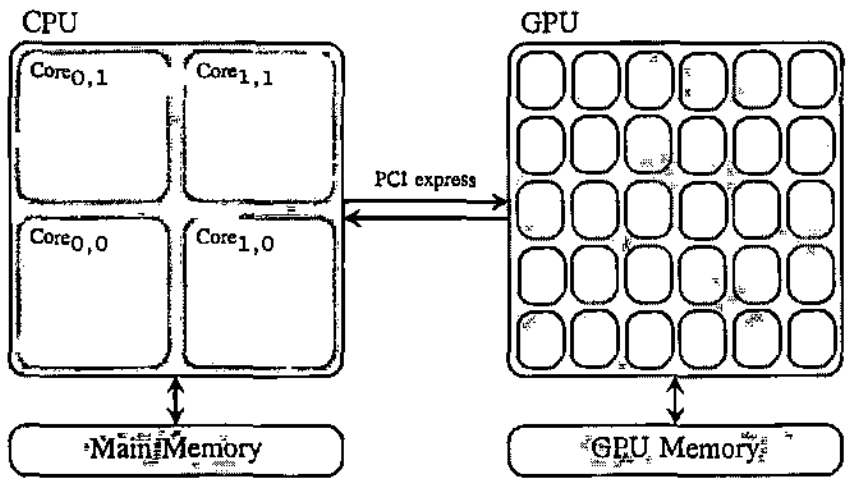


Figura 11: *Estrutura CPU-GPU e sua interface de interconexão PCI Express. Fonte: State-of-the-art in heterogeneaus computing, 2010.*

Como um kernel é chamado a partir de um código serial em uma linguagem de programação convencional, para um host fazer uso da arquitetura heterogênea que suporta CUDA® é necessário apenas que ele tenha uma CPU compatível com a linguagem de programação utilizada e que essa linguagem tenha integração com CUDA® através de um módulo que estende a linguagem. Estes constituem os requisitos lógicos de CUDA®.

Essa arquitetura heterogênea é considerada de fácil implementação e reparo, e também de baixo custo em relação a outras arquiteturas heterogêneas para processamento paralelo de alto desempenho. Qualquer programador com um host capaz de interconectar uma GPU através de um slot PCI-Express pode fazer uso de CUDA® em suas aplicações.

Até 2009, CUDA® não contava com uma IDE para desenvolvimento, o que tornava complicado para detecção de erros e fazia com que novos programadores tivessem bastante dificuldade. Atualmente podemos contar com o NVIDIA® Nexus que é o primeiro ambiente de desenvolvimento especificamente criado para o suporte a CUDA® C, dentre outras linguagens. Esse ambiente de desenvolvimento é integrado ao Microsoft® Visual Studio, facilitando para o programador, pois não terá que migrar para um ambiente de desenvolvimento completamente novo.

> [...] Nexus extende as funcionalidades do Visual Studio oferecendo ferramentas para o gerenciamento de paralelismo massivo, como a habilidade de poder focar e realizar debug em uma única thread sendo executada em paralelo entre centenas, e a possibilidade de visualizar de

> forma simples e eficiente os resultados computados por todas as threads paralelas. (NVIDIA CORPORATION, 2010, p. 19, tradução nossa).

CUDA® ainda pode ser utilizado com mais de uma GPU no mesmo host. O próprio módulo que estende a linguagem oferece suporte à identificação das GPUs no host, possibilitando o gerenciamento de kernels entre as GPUs por parte do programador. Um estudo feito com a utilização de GPGPU distribuída por Harding e Banzhaf (2008) afirma que com a utilização de cluster de computadores com GPUs é possível acelerar a obtenção de resultados na programação de algoritmos genéticos. Logo, datacenters também podem fazer uso de CUDA®, onde teremos um sistema distribuído com vários hosts, e onde cada um deles executará kernels de aplicações remotas em suas próprias GPUs através da rede.

3.0 ASPECTOS DA ARQUITETURA

Ao utilizar um modelo unificado que é integrado como uma extensão de uma linguagem de programação serial, CUDA® torna-se um modelo de GPGPU para paralelização de aplicativos de alto desempenho. Além das estruturas de hardware apresentadas anteriormente, faz-se necessário considerar algumas especificidades deste modelo de programação.

3.1 Interfaces de Programação em CUDA

Para escrever programas CUDA® é necessário utilizar uma interface de programação, a NVIDIA® disponibiliza duas interfaces atualmente: *CUDA C runtime API*, que é mais comumente utilizada por conta do nível de abstração que propõe, ou *CUDA driver API*. A primeira interface é implementada sobre a segunda, isso permite que as duas sejam utilizadas no mesmo código simultaneamente, de acordo com a necessidade do programador.

Podemos interpretar *CUDA C runtime API* e *CUDA driver API* como um modelo em camadas de interfaces de programação para GPUs NVIDIA®, onde o programador pode fazer uso da primeira camada, que prover um nível de abstração sobre a segunda, tornando o código mais limpo e conciso, ou acessar a segunda camada, mais próxima do hardware, fazendo uso de código assembly ou gerenciando código binário.

As duas interfaces oferecem suporte a todas as operações fundamentais em CUDA® como gerenciamento de memória, transferência de dados entre o host e a GPU, gerenciamento de vários dispositivos, etc.

CUDA C runtime API se torna mais simples de ser utilizada, nesta interface o modelo de programação é visto apenas como um conjunto mínimo de extensões para a linguagem de programação C. Utilizando essa interface, os kernels são inseridos no código fonte como funções de C que retornam resultados após o processamento na GPU ser concluído. O *CUDA driver API*, em contrapartida, oferece controle maior sobre a execução do kernel, além de ser independente de qualquer linguagem, desde que a linguagem seja capaz de gerenciar código binário e assembly, ao custo de ser mais difícil para programar e debugar.

Oficialmente a NVIDIA® disponibiliza apenas a *CUDA C runtime API*, que possui a *CUDA driver API* integrada, para CUDA® com C/C++. Outras linguagens oferecem suporte a CUDA®, mas essas APIs são desenvolvidas por terceiros, não ligados diretamente à NVIDIA®, dentre elas podemos destacar as seguintes linguagens e APIs:

- Python - PyCUDA;
- Java - jCUDA;
- .NET - CUDA.NET;
- MATLAB - GPUmat;
- Fortran - FORTRAN CUDA;

Outras linguagens estão sendo testadas e desenvolvidas para dar suporte a CUDA® em um futuro próximo, dentre as linguagens estão: Ruby, PHP, LUA, Perl, Guile e tcl.

CUDA® foi criado e é mantido originalmente pela NVIDIA® e constitui uma plataforma aberta para desenvolvimento de aplicações, contudo uma interface de programação mais promissora está sendo desenvolvida pelo Khronos Group em conjunto com indústrias, companhias e instituições da área de tecnologia, dentre elas a própria NVIDIA®. Da mesma forma que o grupo desenvolveu o OpenGL®, plataforma gráfica aberta, estão propondo o OpenCL® que pode ser entendida como:

> [...] um padrão multivendedor para programação paralela de uso genérico em sistemas heterogêneos que incluem CPUs, GPUs e outros processadores. OpenCL® disponibiliza um ambiente de desenvolvimento uniforme para os desenvolvedores de software escreverem códigos eficientes e portáveis para computação de alta performance em servidores, desktops e dispositivos handheld. (KHRONOS GROUP, 2009, p. 1, tradução nossa).

OpenCL® é um framework para programação de sistemas heterogêneos e já tem uma aceitação considerável entre os programadores que utilizam CUDA®. Da mesma forma que outras interfaces de programação aberta ganharam destaque, com o auxílio de grandes empresas e instituições, OpenCL® está sendo desenvolvido com o mesmo objetivo promissor de unificar CUDA® com outras arquiteturas GPGPU em um mesmo modelo de programação.

3.2 Portabilidade e Escalabilidade

CUDA C foi a API que impulsionou o GPGPU em seu início, até que ele fosse percebido como uma altemativa em processamento paralelo. Através do seu nível de abstração, controle de gerenciamento recursos transparente ao programador e provendo todos os recursos necessários para a programação de GPUs NVIDIA® tornou-se um modelo de interface de programação genérica em GPUs. Atualmente outras APIs estão surgindo e o OpenCL® está ganhando destaque como uma interface aberta que pretende unificar todas as plataformas heterogêneas de processamento.

Códigos em CUDA C possuem um alto grau de portabilidade entre aplicações que utilizam esta API. Como um kernel tem comportamento similar a uma função dentro do código serial da aplicação, ele pode ser levado à outra aplicação do mesmo modo que funções em linguagens de programação serial. Após essa migração, cabe ao programador analisar os recursos de hardware disponíveis para a execução desse kernel e modificar a alocação de recursos em tempo de execução, como grids e thread blocks, para a obtenção do melhor desempenho.

A API aberta pretende prover esse nível de portabilidade para um conjunto de dispositivos de fabricantes diferentes. Assim, algumas modificações são feitas em relação ao modelo de programação proposto por cada fabricante, como CUDA C, por exemplo, mas, como afirma Komatsu (2010) estes modelos têm hierarquia de memória e elementos de processamento similares.

Rul et al. (2010) afirma que devido a similaridade na estrutura do programa entre as linguagens CUDA® e OpenCL®, o processo de tradução é relativamente direto". A portabilidade entre estas APIs torna-se um processo simplificado.

Algumas diferenças entre os modelos devem ficar bem claras ao programador para que o processo de portabilidade ocorra de maneira direta e simples, por exemplo:

> Em OpenCL, um NDRange é espaço indexado em N dimensões, correspondendo ao grid em CUDA. Um work-group e um work-item correspondem a um thread-block e uma thread em CUDA, respectivamente. Então, o modelo de execução no OpenCL é similar ao de CUDA. (KOMATSU, 2010, p. 4, tradução nossa).

Podemos perceber que nesta comparação não existe um elemento em OpenCL correspondente a warp do modelo CUDA C. Essas diferenças, entre o OpenCL e outras interfaces proprietárias de programação, ignorando características particulares de cada uma, é que permitem a portabilidade de código de várias arquiteturas heterogêneas para o OpenCL.

OpenCL prover um outro tipo de abstração de hardware em relação as plataformas proprietárias de GPGPU, por isso requer um método de programação diferente em relação a CUDA C, sendo necessárias abordagens diferentes para otimização e aquisição de performance máxima. Como conclusão de um estudo feito sobre a performance entre CUDA C e OpenCL, Komatsu et at. apresenta:

> A avaliação quantitativa de resultados indica que o desempenho atingido de todos os programas em OpenCL com o modo padrão é bem menor que aquela equivalente em um programa CUDA. [...] a diferença de performance vem da diferença das capacidades de otimização entre os compiladores. Otimizando manualmente os códigos dos kernels dos programas OpenCL e/ou usando otimizações automáticas no compilador C

> OpenCL, a performance alcançada torna-se quase a mesma performance de CUDA. (KOMATSU et al., 2010, p. 14, tradução nossa).

Além de prover um método simples de portabilidade de código, CUDA® e OpenCL® ainda dispõem de características intrínsecas do modelo que os tornam propícios ao escalonamento.

CUDA® por si só já é uma plataforma escalonável através dos múltiplos processadores existentes em uma GPU. Escalabilidade é uma condição nativa nas interfaces de programação para GPGPU. Em CUDA® essa propriedade é transparente ao programador, os controladores da placa fazem o trabalho de gerenciar o escalonamento das warps através dos SMs.

Através da implementação nativa de funções que gerenciam mais de uma GPU no mesmo host, fica evidente a facilidade em escalonar um sistema a nível de GPUs. Esse tipo de escalonamento requer um controle do programador sobre o que será executado em cada GPU, o que pode tornar o código susceptível a erros, pois não há uma sincronização garantida entre as GPUs podendo surgir, até mesmo, deadlocks.

Outras experiências com CUDA® demostram a possibilidade de utilizar GPGPU como parte de um sistema distribuído. Harding et al. (2008) demonstra em uma experiência como é possível utilizar GPUs distribuídas em vários hosts como parte de um sistema distribuído destacando que neste trabalho, uma biblioteca CUDA.NET foi usada. Esta biblioteca permite que qualquer linguagem .NET se comunique com o CUDA driver. Do mesmo modo que podemos distribuir a execução utilizando uma linguagem .NET, isso pode ser feito com qualquer outra linguagem que tenha suporte a CUDA C.

3.3 Restrições e Otimização

Em qualquer modelo de programação para processamento de alta performance a otimização de código é um fator determinante. Cada arquitetura possui restrições e fatores intrínsecos que podem reduzir o ganho esperado em tempo de processamento. Um estudo desses fatores e aplicação de técnicas específicas à arquitetura utilizada pode melhorar o desempenho da aplicação.

De forma geral, como apresentado no guia para otimização de aplicações em CUDA para arquitetura Fermi® por NVIDIA CORPORATION (2010), a otimização em CUDA consiste em algumas recomendações prioritárias:

- Achar uma maneira de paralelizar o código serial;

- Minimizar a transferência de dados entre o host e o dispositivo;
- Ajustar a configuração de chamada aos kernels para maximizar a utilização do dispositivo;
- Assegurar que o acesso à memória global é coalescente;
- Trocar o acesso à memória global por um acesso a memória compartilhada sempre que for possível;
- Evitar caminhos de execução diferentes no mesmo warp.

Paralelizar o código e saber que parte do código serial deve ser executada na GPU é fundamental quando se trabalha em sistemas heterogêneos com processamento paralelo SIMT. Em uma aplicação serial, apenas uma porção do código pode ser paralelizado, ou mesmo nenhuma, dependendo da natureza da aplicação, a fim de alcançar mais desempenho em sistemas SIMT. O programador deve estar ciente de como funciona a arquitetura e seu modelo de processamento paralelo para determinar as partes do código passíveis de execução paralela.

A GPU é conectada ao host através de um slot de expansão PCI-Express x16 que pode constituir um gargalo de dados entre o host e a GPU. Isso é uma limitação tecnológica que até o final deste ano deve ser minimizada, com o lançamento da terceira versão do PCI-Express. Como esse barramento constitui o único meio de acesso a GPU é necessário saber usá-lo eficientemente, através do gerenciamento inteligente da transferência de dados entre o host e o dispositivo. O programador deve evitar transferência excessiva de dados para GPU, reduzindo o efeito do gargalo de dados no barramento.

> Uma maneira de conseguir isso é mover a maior quantidade de código do host para o dispositivo, mesmo que isso signifique executar kernels com baixo paralelismo. Estruturas de dados intermediárias devem ser criadas na memória do dispositivo, operadas pelo próprio dispositivo, e destruídas sem mesmo terem sido mapeadas pelo host ou copiadas para a memória do host. (NVIDIA CORPORATION, 2010, p. 87, tradução nossa).

Essa solução é a melhor possível, pois o impacto na performance de uma grande transferência de dados é menor em relação a várias transferências pequenas de dados. Outra maneira de amenizar esse problema é utilizar transferência de dados em blocos através de DMA.

Segundo Carrillo, Siegel e Li (2008), uma das tarefas principais na otimização de um programa para CUDA® é encontrar um número ótimo de threads e thread blocks que irão manter a GPU totalmente ocupada. Essa constitui uma das tarefas mais desafiadoras na

otimização de código. Várias soluções vêm sendo propostas para atingir um uso eficiente do dispositivo tais como:

- modelos que avaliam o consumo de energia em relação a performance, (HONG; KIM, 2009);
- utilização de um nível de abstração maior sobre CUDA®, (HAN; ABDELRAHMAN, 2008);
- análise de estruturas de controle e propostas de aperfeiçoamento de uso, (CARRILO; SIEGEL; LI, 2008);

Dentre outras soluções particulares propostas pela comunidade que faz uso de CUDA®, a NVIDIA® "disponibiliza uma ferramenta chamada occupancy calculator que permite ao programador facilmente calcular o melhor tamanho do blodo de thread baseado no uso do registro e da memória compartilhada em um kernel" (JANG et al., 2009, p. 187, tradução nossa).

Estruturas de controle como *if-else* que podem alterar o fluxo de execução do programa devem ser evitadas, segundo Ryoo (2008) porque a concorrência entre as threads será reduzida se as threads do mesmo grupo de threads (chamado warp em CUDA®) seguirem diferentes contextos. Essa mudança de fluxo dentro da mesma warp vai fazer com que o uso da GPU seja depreciado.

O uso da memória global também constitui uma restrição importantíssima nesta arquitetura, como afirma Ryoo et al. (2008, p. 5, tradução nossa) "[...] a largura de banda da memória global pode limitar o throughput do sistema. Aumentar a quantidade de threads não ajuda neste tipo de situação." Os problemas quanto ao uso de memória global são os problemas mais complicados para solucionar, pois não possuem métodos fixos, e gerenciar uma maneira de esconder a latência da memória global é uma atividade comum aos programadores em CUDA®. Cada implementação de programas CUDA® requer uma solução diferente para esse problema, baseado na utilização de recursos do dispositivo na aplicação considerada. "Balancear o uso destes recursos é não intuitivo e algumas aplicações podem atingir o limite de recursos além de problemas com instruções [...]" (RYOO et al., 2008, p. 77, tradução nossa).

O acesso coalescente à memória é algo complexo e solicita do programador um conhecimento avançado da hierarquia de memória do dispositivo. Esse gerenciamento deve ser feito pelo programador de forma direta durante a alocação de memória utilizada pelo programa. Vejamos a consideração de um grupo de pesquisadores, que desenvolveu uma

ferramenta para ser utilizada com CUDA que faz esse gerenciamento de memória coalescente de forma automática:

> Considerando que o comportamento da memória coalescente é atualmente de entendimento complexo, nós acreditamos que estas transformações são melhor suportadas por uma ferramenta automática. Isso reduz o potencial para erros na escrita de códigos para memória coalescente [...] (UENG et al., 2009, p. 6, tradução nossa).

CUDA® ainda possui uma restrição muito importante, não suportar recursividade. É importante perceber que um kernel só pode ser chamado pelo host, não há como fazer a chamada de um kernel dentro de outro que está sendo executado, nem é possível fazê-lo através da GPU. O host, como a própria definição sugere, controla todo o fluxo de dados e instruções processadas na GPU.

Na arquitetura Fermi há um grande esforço por parte da NVIDIA® para reduzir todas essas restrições, principalmente no que toca a hierarquia de memória e trazendo, nessa geração nova, um modelo diferenciado das anteriores.

4.0 APLICABILIDADE E DESEMPENHO

O modelo unificado da NVIDIA®, por ser pioneiro como um modelo de programação genérica de alto desempenho com um nível de abstração de hardware nunca visto nessa área, não tivera, de início, toda a credibilidade que conquistou nos dois últimos anos. Desde o seu lançamento em 2005 os programadores, pesquisadores, acadêmicos e companhias vêm buscando soluções em processamento que podem explorar o potencial de CUDA® nas mais diversas áreas e com esses esforços a plataforma ganha aceitação e sendo aperfeiçoada.

CUDA® é uma plataforma de alto desempenho, mas sua performance depende bastante do conhecimento que o programador detém sobre a arquitetura e condições intrínsecas ao programa que podem ir de encontro as restrições da plataforma, resultando em um ganho de desempenho menor que a média. Contudo, se for utilizada em uma aplicação onde o modelo de processamento de dados for adequado ao SIMT, CUDA® pode atingir ganhos de performance de até 1800 vezes, se comparado com a mesma aplicação sendo executada em uma CPU.

CUDA® é o primeiro modelo de programação GPGPU a ser utilizado em áreas como: finanças, ciências biológicas, automação de projetos eletrônicos e estudos relativos a petróleo e gás. Além dessas áreas existem trabalhos com CUDA® em ferramentas de programação, dinâmica dos fluidos, física dos jogos, gráficos, áudio, imagem, imagens médicas, processamento de sinais, aplicações matemáticas, dentre outras. Algumas aplicações recentes de CUDA® nas mais diversas áreas foram selecionadas para serem apresentadas e analisadas nesse trabalho.

4.1 Remodelagem de Framework Baseado em Agentes

Sistemas de simulação baseados em agentes são utilizados em vários eixos da ciência atualmente, principalmente nas ciências biológicas e na computação através dos algoritmos genéticos para simulação de populações. O trabalho de Richmond, Coakley e Romano (2009) descreve os aspectos principais da adaptação do framework para CUDA. O trabalho foi implementado sobre o *FLAME (Flexible Agent Modelling Environment)*, o mesmo utilizado para a versão do framework em CPUs. No trabalho são demonstradas soluções para latência de acesso à memória global, segundo Richmond, Coakley e Romano (2009) cada thread na GPU representa um único agente, assim o uso da memória global entre agentes paralelos é evitado. Ainda é destacado que:

> A chave para uma comunicação de mensagens eficiente entre agentes $O(n^2)$ é otimizar o uso da memória compartilhada (SM). Isto permite que as mensagens sejam paginadas dentro dos SMs e processadas serialmente por cada thread no processador, com um acesso rápido a um registro. (RICHMOND; COAKLEY; ROMANO, 2009, p. 2, tradução nossa).

Estes autores fizeram duas simulações para análise de resultados obtidos. Na primeira simulação é utilizado um único tipo agente, uma mensagem de localização e três funções para o agente. Na segunda são utilizados dois tipos de agentes, três tipos de mensagens e seis funções diferentes. O resultado na performance foi apresentado na forma de um gráfico, aqui apresentado na Figura 12, que mostra o ganho de desempenho para as duas simulações em relação ao tamanho da população de agentes.

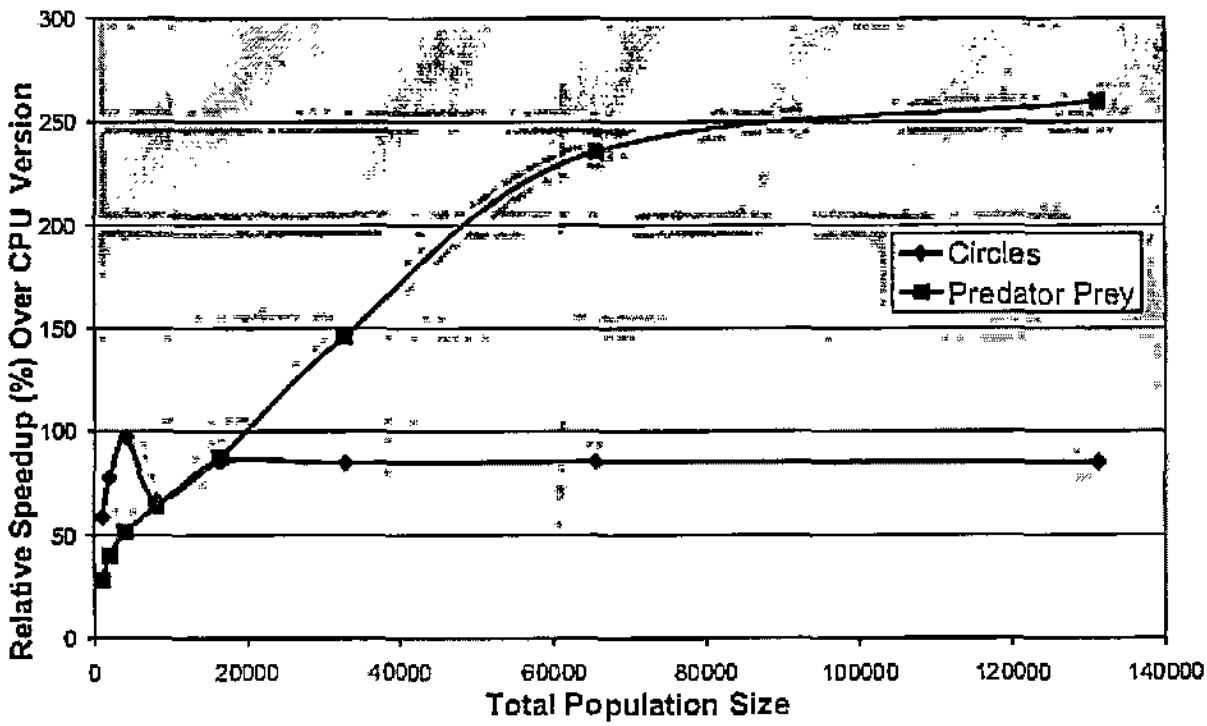


Figura 12: *Gráfica comparativo em percentual entre o desempenho de uma simulação de agentes entre uma implementação em CPU (linha inferior cam círculos) e outra em GPU utilizando CUDA® (linha superior com quadrados). Fante: A High Performance Agent Based Modelling Framework on Graphics Card Hardware with CUDA (Extended Abstract), 2009.*

É possível perceber com a análise do gráfico um ganho de mais de 50% em relação a versão para CPU na primeira simulação, com um único tipo de agente. Na segunda simulação temos um ganho de mais de 250%. A segunda simulação tem um ganho de performance maior em relação a primeira devido ao grau de paralelização exigido no segundo caso, fazendo os kernels bem mais eficientes em relação ao código serial.

4.2 Aceleração de Operações em Bancos de Dados SQL

A performance de um banco de dados, em uma aplicação que faz uso constante de dados armazenados, é um fator determinante na performance do aplicativo como um todo, constituindo elemento crítico e determinante ao desempenho. Alguns esforços na aceleração de transações em bancos de dados com GPU já existiam, mas segundo Bakkum e Skadron (2010) os primeiros trabalhos mostram acelerações drásticas em várias operações de bancos de dados, mas fazendo uso de primitivas que não fazem parte de linguagens de banco de dados convencionais como SQL. No trabalho Bakkum e Skadron (2010) é apresentada uma solução em desempenho para banco de dados usando a linguagem SQL. Nesse trabalho os pesquisadores utilizam o SQLite como base, transcrevendo a parte do aplicativo que faz as consultas ao banco de dados – rotinas SELECT – para ser executado na GPU sem a necessidade de mudança na linguagem SQL.

Bakkum e Skadron (2010) se referem à arquitetura do SQLite como relativamente simples, onde o núcleo da infraestrutura do SQLite contém uma interface com o usuário, o processador de comandos SQL e uma máquina virtual além de outros componentes menos relevantes.

> O ponto crucial desse projeto é a reimplementação da máquina virtual do SQLite com CUDA. A máquina virtual é implementada como um kernel CUDA que executa os procedimentos opcode. O projeto implementou em torno de 40 opcodes [...]. (BAKKUM; SKADRON, 2010, p. 98, tradução nossa).

Opcodes executam operações específicas no banco de dados, como abrir uma tabela, carregar dados, realizar operações matemáticas em um registro e pode pular para outro opcode. Esse códigos no SQLite para CPU são executados um a um após o processador de comandos gerar um programa intermediário que utilizará os opcodes.

A maior restrição imposta pela arquitetura CUDA foi a quantidade de memória disponível nas GPUs, que atualmente chegam ao limite de 4GB. Essa limitação faz com que não seja possível trabalhar com consultas que retomem um grande volume de dados. O gargalo existente entre o host e a GPU (barramento PCI-Express) constitui outra limitação para o projeto. Para minimizar a influência dessas restrições no trabalho, foi adotada uma medida de controle, que consiste em carregar uma parte do banco de dados para a GPU e realizar múltiplas consultas sobre esse conjunto de dados. Além disso, o uso da memória global para armazenar os dados que serão consultados constitui outro fator limitador da performance no projeto.

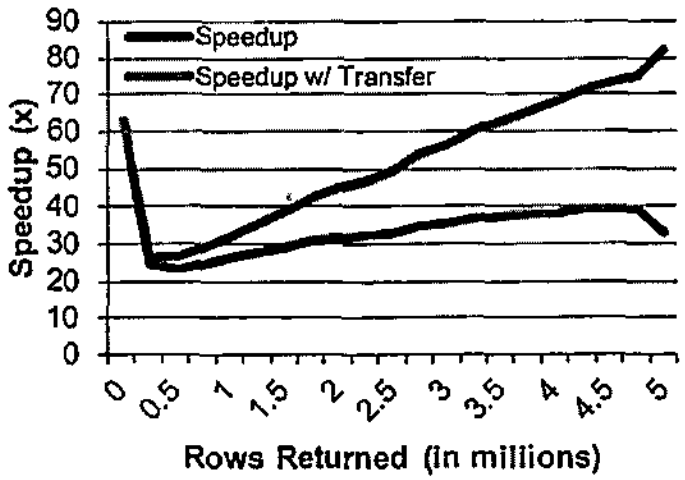


Figura 13: *Gráfico representativo do ganho de desempenho para uma consulta em um banco de dados baseado em SQL utilizando CUDA em relação ao número de dados consultados em milhões. A linha superior representa o ganho de performance na consulta, e a linha inferior o ganho de desempenho da consulta com o retorno dos dados ao host. Fonte: Acceierating SQL Database Operations on a GPU with CUDA, 2010.*

Com essas medidas de controle principais foi possível atingir alguns resultados positivos em testes com treze tipos de queries diferentes, como apresentado na figura 13. "O aumento de velocidade médio ao longo de todas as queries foi de 50X, reduzido a 36X quando o tempo de transferência dos resultados é incluído" (BAKKUM; SKADRON, 2010 – p. 6, tradução nossa).

4.3 Acelerando Codificação de Vídeo

A codificação de vídeo é conhecida como um processo intenso de processamento e com as imagens em alta definição, com resoluções de vídeo altíssimas, é necessário à busca por novas soluções a fim de tornar esse processo mais rápido. Uma das técnicas utilizadas em processamento de vídeos é o Motion Estimation, nessa técnica partes de um frame são selecionadas e analisadas através de um algoritmo de soma das diferenças absolutas (SAD) e comparado com partes do frame anterior, a fim de encontrar semelhanças ou diferenças mínimas. No trabalho de Colic, Kalva e Furht (2010) são demonstrados resultados de análise de desempenho com diferentes níveis de otimização do processo de Motion Estimation e utilização de três GPUs diferentes. Na primeira parte do trabalho foi necessário definir como o código serial seria paralelizado, nesse ponto foram consideradas quatro possibilidades, das quais duas foram utilizadas no experimento após um estudo sobre as limitações de cada método.

O experimento para análise de resultados foi dividido em duas partes. Na primeira parte é feita uma análise sobre os resultados de diferentes níveis de otimização de código aplicados sobre a mesma GPU, uma 8800GTX. São definidos 4 níveis de otimização e código serial para servir de base:

- *Baseline Code* – todo implementado em C e executado somente na CPU;
- *Only Global Memory* – Código executado na GPU, mas sem nenhuma otimização;
- *Temporary Variable for Accumulation* – Nessa implementação são utilizadas variáveis temporárias para as threads que acumulam os resultados de calcúlos da SAD;
- *Shared Memory* – Nesse ponto é introduzido o uso da memória compartilhada, mas sem nenhuma otimização relacionada;
- *Fully Optimized Code* – Código completamente otimizado.

Para estabelecer estes níveis de otimização e a versão completamente otimizada Colic, Kalva e Furht (2010) consideram que para atingir uma performance máxima em uma aplicação para GPU devemos focar nos seguintes objetivos: remover leitura de dados não-coalescente na memória global, fazer uso da memória compartilhada à memória global, remover conflitos de memória compartilhada evitando a execução de threads em série e o uso de blocos de thread com tamanhos múltiplos de 16. Nas Figuras 14, 15 e 16 a seguir são demonstrados resultados da primeira parte do experimento fazendo uso de três resoluções diferentes de vídeo e demonstrando a performance de cada passo da otimização.

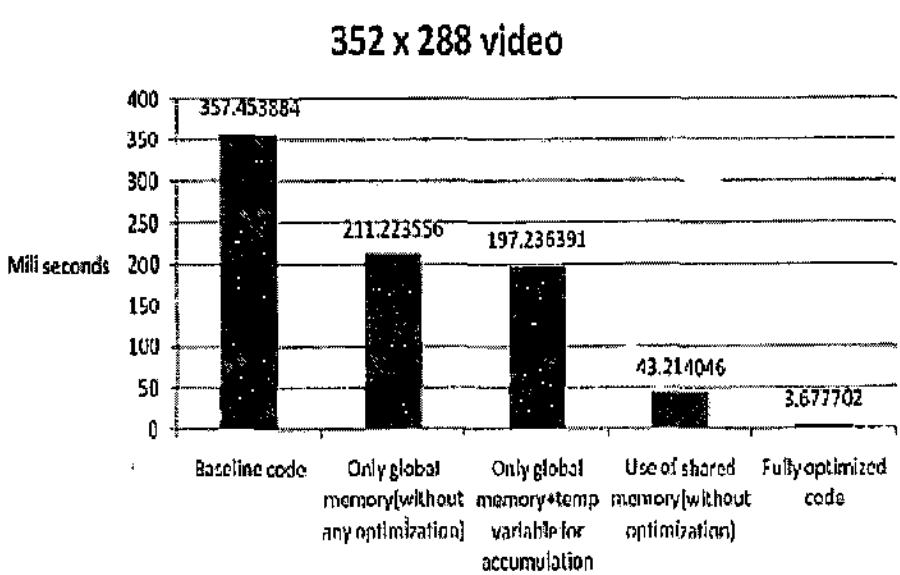


Figura 14: *Medidas de tempa em milissegundas para renderização de vídeos com a resolução de 352 x 288 pixels com todos os passo de otimização apresentados do esquerda para a direita, respectivamente. Fonte: . Exploring NVIDIA-CUDA for Video Coding, 2010.*

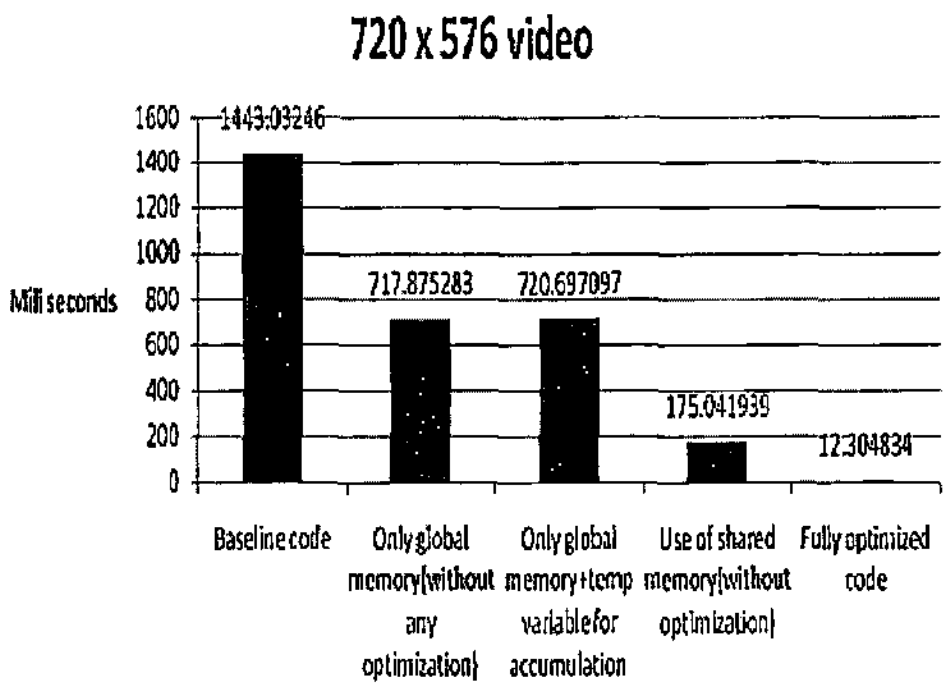


Figura 15: *Medidas de tempo em milissegundos para renderização de vídeos com a resolução de 720 x 576 pixels com todos os passo de atimização apresentados da esquerda para a direita, respectivamente. Fonte: . Exploring NVIDIA-CUDA far Video Cading, 2010.*

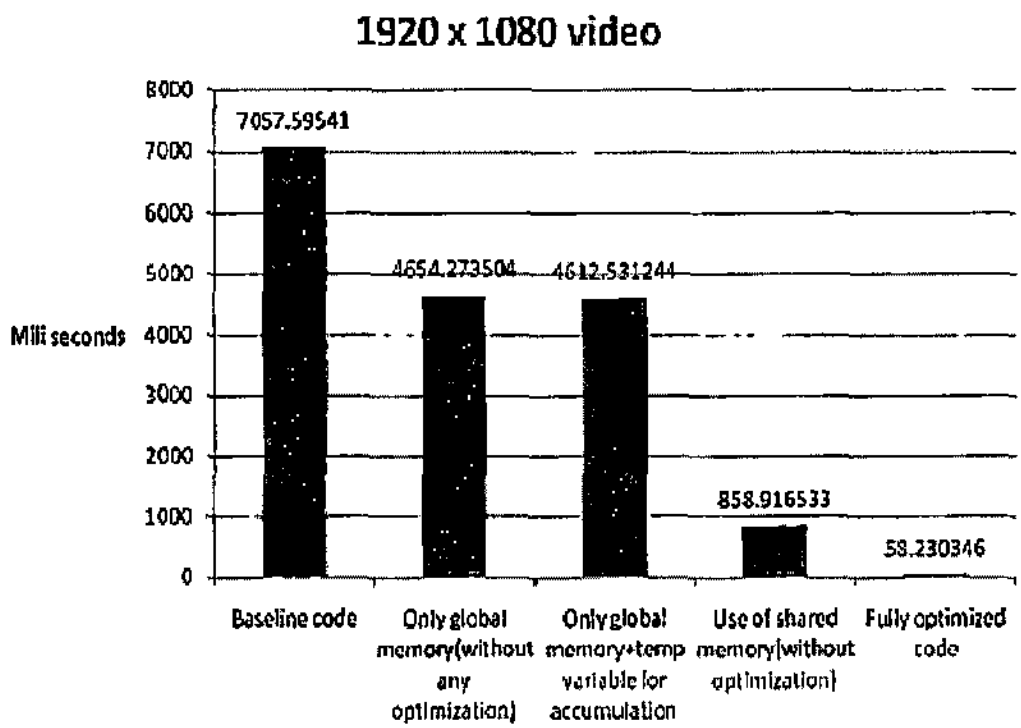


Figura 16: *Medidas de tempo em milissegundos para renderização de vídeos com a resolução de 1920 x 1080 pixels com todos os passo de otimização apresentados da esquerda para a direita, respectivamente. Fonte: . Explaring NVIDIA-CUDA for Video Coding, 2010.*

Na segunda parte do experimento são feitas análises da escalabilidade de CUDA através da execução do mesmo algoritmo em hardwares diferentes. Os hardwares utilizados são 8800GT com 14 SMs, 9800GTX com 16 SMs e GTX285 com 30 SMs. No trabalho são apresentados gráficos detalhados sobre resultados intermediários da segunda parte experimental. A seguir são apresentados gráficos relativos ao aumento de performance com cada placa e no final um comparativo de performance entre elas.

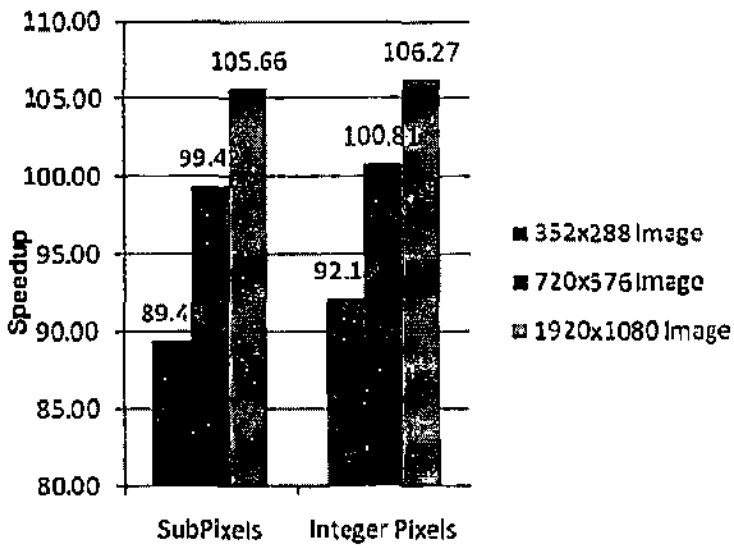


Figura 17: *Comparativo de aumento de performance com o aumento de resolução em uma GPU 8800GT. Fonte: . Exploring NVIDIA-CUDA far Video Coding, 2010.*

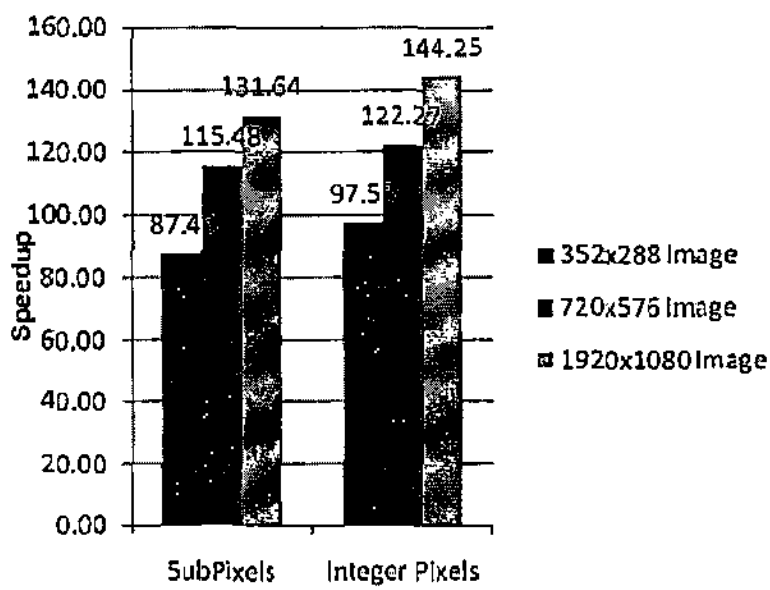


Figura 18: *Comparativo de aumento de performance com o aumento de resolução em uma GPU 9800GTX. Fonte: . Exploring NVIDIA-CUDA for Video Coding, 2010.*

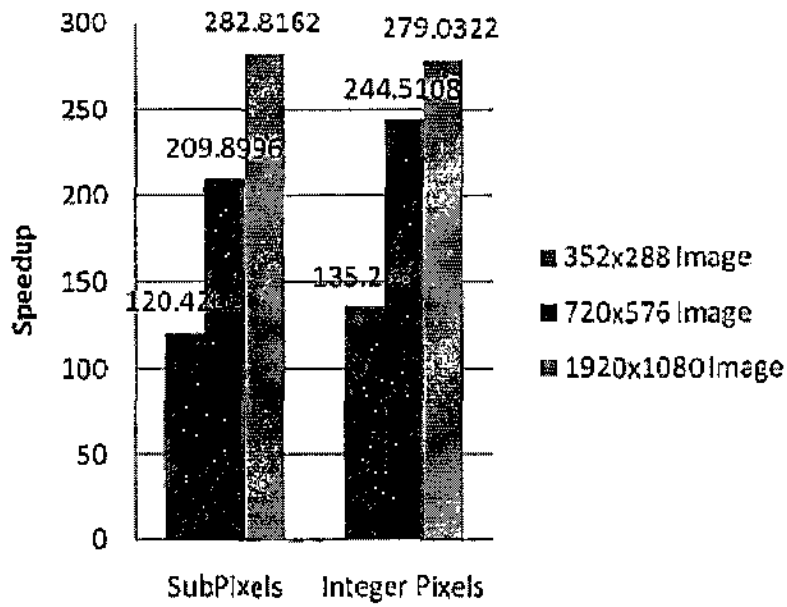


Figura 17: *Comparativo de aumento de performance com o aumento de resolução em uma GPU 285GTX. Fonte: . Exploring NVIDIA-CUDA for Video Coding, 2010.*

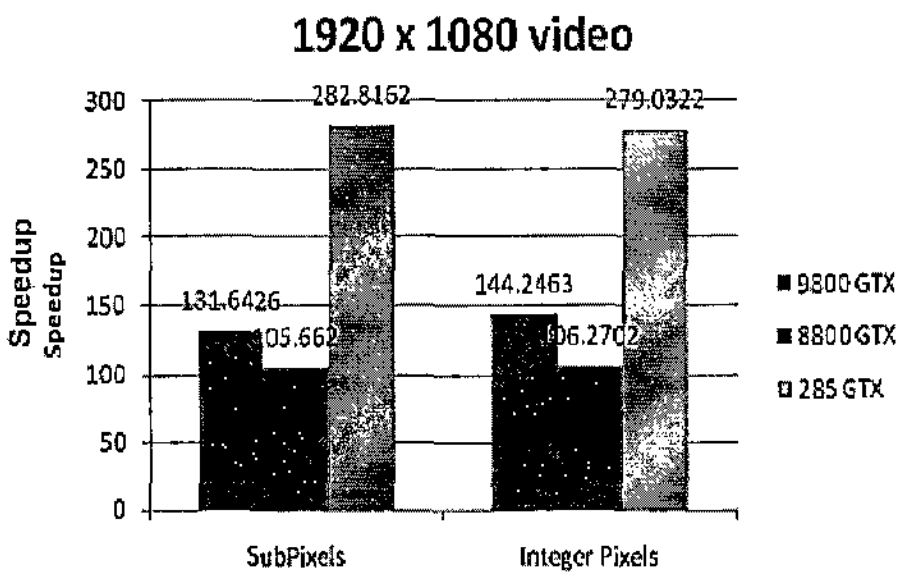


Figura 17: *Comparativo de aumento de performance em resolução de 1920 X 1080 entre as GPUs 8800GT, 9800 GTX e 285. Fonte: . Exploring NVIDIA-CUDA for Video Coding, 2010.*

Através da análise dos gráficos fica perceptível o quanto a arquitetura CUDA vem evoluindo e proporcionando melhor desempenho em aplicações práticas. Mesmo utilizando

uma placa da primeira geração CUDA é possível obter uma performance 100 vezes melhor, em relação à mesma aplicação para CPUs. Com uma placa da penúltima geração CUDA é visto um processamento paralelo, em média, 280 vezes mais rápido em relação a uma CPU.

5 CONSIDERAÇOES FINAIS

A indústria de processadores atual aposta em um paradigma de processamento multicore que ainda está longe de ser considerado um padrão, pois não existem linguagens de uso comercial que suportem os recursos oferecidos por esses núcleos de processamento. Para padronizar os processadores multicore para CPUs desenvolvidos atualmente seria necessária uma mudança radical em todo o modelo de programação estruturada, linguagens de programação, APIs, ambientes de programação, sistemas operacionais entre outros, seria como remodelar todo o processamento de dados existente.

A busca por uma solução em processamento de dados que mantenha as necessidades globais supridas torna-se crítica e o GPGPU se destaca como uma solução de baixo custo, fácil implementação e acessível em termos de hardware e software. GPGPU, como todas as outras soluções em processamento paralelo, em seu início fora complicado para implementação por conta da própria natureza do hardware que faz uso. A introdução de processadores de fluxo de uso genérico em GPUs possibilitou que esta fosse utilizada para o GPGPU. O modelo de processamento paralelo de alto desempenho CUDA® da NVIDIA® vai ao encontro dessas novas possibilidades de hardware e proporciona ao programador a possibilidade de explorar esse modelo híbrido de forma direta e transpor barreiras que são inerentes ao processamento paralelo através de um modelo de processamento e hardware de alta performance bem estruturado que torna transparente alguns aspectos complexos da programação paralela.

CUDA® a cada geração amadurece e se firma como um modelo de GPGPU que já serve de base para outros em desenvolvimento. Do ponto de vista do programador, CUDA® não exige uma mudança radical de paradigma de programação e integra-se facilmente às linguagens existentes, facilitando a portabilidade de código e aplicações. Desde seu lançamento, CUDA® é utilizado em diversas áreas onde são exigidas performances ótimas em processamento de dados proporcionando aumentos de desempenhos muito superiores às CPUs multicore.

Como todo modelo de processamento de dados CUDA® apresenta algumas restrições que o impedem de atingir ganhos maiores de desempenho em certas aplicações, mas com a tendência de aperfeiçoamento que vêm experimentando a cada geração e aprimoramento do hardware, essas restrições devem ser minimizadas.

Através da análise de trabalhos em CUDA® é possível perceber que ele oferece uma solução que supre a necessidade por processamento de alto desempenho, sem exigir uma

remodelagem radical no processamento de dados. O modelo unificado da NVIDIA® ainda apresenta-se como um modelo de baixo consumo de energia, que é uma característica que deve predominar nos novos sistemas de processamento de dados de alto desempenho.

Acredito que kernels podem ser usados como parte integrante da biblioteca de uma linguagem de programação com suporte a CUDA®, de modo que o kernel possa ser chamado em detrimento à função serial da biblioteca, desde que o host possua uma GPU com suporte a CUDA®. Em trabalhos futuros serão feitas análises e utilização prática de CUDA® em aplicações que podem ser altamente paralelizadas na solução de problemas que requerem processamento intenso de dados como: algoritmos genéticos, pesquisa e ordenação bem como a integração dessas funções em bibliotecas de linguagens de programação como Python.

REFERÊNCIAS BIBLIOGRÁFICAS

BAKKUM P.; SKADRON K. **Accelerating SQL Database Operations on a GPU with CUDA**, Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, Pittsburgh, Pennsylvania, EUA: ACM, 2010, p. 64-103.

BOYD, C. **Data-Parallel Computing**, ACM QUEUE, Vol. 6, No. 2, New York, NY, USA: ACM 2008, p. 30-38.

BRODTKORB, A. et al. **State-of-the-Art in Heterogeneous Computing**, SINTEF ICT, Department of Applied Mathematics, Oslon, Noruega: IOS Press, 2010. 33 p. Disponível em: <http://babrodtk.at.ifi.uio.no/files/publications/brodtkorb_etal_star_heterocomp_final.pdf>. Acesso em: 20 de Julho de 2010.

BYDAL, Asbjorn. **Implementation of Genetic Algorithm on CUDA**, IKT502 Project Report, University of Agder, Noruega: 2008. 16 p. Disponível em: <http://fruit.grm.hia.no/ikt502/year2008/projects/reports/Implementation_of_genetic_algorithm_on_CUDA.pdf>. Acesso em: 20 de Julho de 2010.

CARRILLO, S; Siegel, J.; Li, X. **A Control-Structure Splitting Optimization for GPGPU**, Proceedings of the 6th ACM Conference on Computing Frontiers, Ischia, Italia: 2009, p. 147-150.

COLIC, A.; KALVA, H.; FURHT, B. **Exploring NVIDIA-CUDA for Video Coding**, Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems, Phoenix, Arizona, EUA: ACM, 2010, p. 13-22.

CREEGER, M. **Multicore CPUs for the Masses**, ACM QUEUE, Vol. 3, No. 7, New York, NY, EUA: ACM, 2005, p. 63-64.

FATAHALIAN, Kayvon; HOUSTON, Mike. **GPUs a Closer Look**, ACM QUEUE, Vol. 6, No. 2, New York, NY, EUA: 2008, p. 19-28.

HAN, T. D.; ABDELRAHMAN, T. S. ***hi*CUDA: A High-level Directive-based Language for GPU Programming**, Proceedings of 2nd Workshop on General Purpose Processing on raphics Processing Unit, Washington, DF, EUA: 2009, p.52-61.

HARDING, S.; BANZHAF, W. **Distributed Genetic Programming on GPUs Using CUDA**. In: Workshop on Parallel Architectures and Bioinspired Algorithms, Raleigh, USA: 2009. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.158.5783&rep=rep1&type=pdf>. Acesso em: 23 de Julho de 2010.

HENNESSY, John; PATTERSON, David. **A Conversation with John Hennessy and David Patterson: They Wrote the Book of Computing**, ACM QUEUE, Vol. 4, No. 10, New York, NY, EUA: ACM: 2006, p. 14-22, Entrevista concedida a Kunle Olukotun.

HONG S.; KIM, H. **An Integrated GPU Power and Performance Model**, International Symposium on Computer Architecture, Saint-Malo, França: 2010, p. 280-289.

JANG et al. **Multi GPU Implementation of Iterative Tomographic Reconstruction Algorithms,** Proceedings of Sixth IEEE Internacional Conference on Symposium on Biomedical Imaging: From Nano to Macro, Boston, Massachusetts, EUA: IEEE Press, 2009, p. 185-188.

KHRONOS GROUP, **OpenCL API 1.0 Quick Reference Guide,** Ver. 1109, 2009. 6 p. Disponível em: <http://www.khronos.org/files/opencl-quick-reference-card.pdf>. Acesso em: 25 de Agosto de 2010.

KIRK, David; HWU, Wen-mei. **Programming Massively Parallel Processors: A Hands-on Approuch,** versão para referências em cursos, disponível em: <http://courses.engr.illinois.edu/ece498/al/Syllabus.html>. Acesso em: 14 de Julho de 2010.

KOGGE, P. et al. **ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems,** DARPA IPTO, Technical Report, EUA: DARPA 2008. 278p. Disponível em: <http://www.er.doe.gov/ascr/Research/CS/DARPA%20exascale%20-%20hardware%20(2008).pdf>. Acesso em: 26/08/2010

KOMATSU K. et al. **Evaluating Performance and Portability of OpenCL Programs.** In: International Workshop on Automatic Performance Tuning, Berkeley, CA, EUA: 2010. Disponível em: < http://vecpar.fe.up.pt/2010/workshops-iWAPT/Komatsu-Sato-Arai-Koyama-Takizawa-Kobayashi.pdf>. Acesso em: 25 de Agosto de 2010.

NICKOLLS, J. et al. **Scalable Parallel Programming with CUDA,** ACM QUEUE, Vol. 6 No. 2, New York, NY, EUA: ACM 2008, p. 40-53.

NVIDIA CORPORATION, **NVIDIA's Next Generation CUDA Compute Architecture: Fermi,** V. 1.1, Santa Clara, CA, EUA: 2009. 21 p. Disponível em: < http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitecture Whitepaper.pdf>. Acesso em: 10 de Julho de 2010.

_____, **NVIDIA CUDA C Programming Guide,** V. 3.1, Santa Clara, CA, EUA: 2010. 161 p. Disponível em: <http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ ProgrammingGuide_3.1.pdf>. Acesso em: 01 de Julho de 2010.

_____, **Tuning CUDA Applications for Fermi,** V 1.0, Santa Clara, CA, EUA: 2010. 6 p. Disponível em: < http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_FermiTuning Guide.pdf>. Acesso em: 01 de Julho de 2010.

OWENS, John D. et al. **A Survey of General-Purpose Computation on Graphics Hardware,** Computer Graphics Forum, Vol. 26, No. 1: 2007, p. 80-113.

PARHAMI, B. **Introducing to Parallel Processing Algorithms and Arehitectures,** New York: Kluwer Academic Publishers, 2002.

PATTERSON. D. **The Trouble with Multi-Core,** IEEE Spectrum, Vol. 47, No. 7, Internacional, EUA: IEEE Press, 2010, p. 24-29.

PENG, Lu et al. **Memory Performance and Scalability of Intel's and AMD's Dual-Core Processors: A Case Study**, Performance, Computing, and Communications Conference, New Orleans, LA, EUA: 2007, p. 55-64.

RICHMOND, P.; COAKLEY S.; ROMANO D. M. **A High Performance Agent Based Modelling Framework on Graphics Card Hardware with CUDA (Extended Abstract)**, Proceedings of 8th International Conference on Autonomous Agents and Multiagent Systems, Budapest, Hungria: IFAAMAS, 2009, p. 1125-1126.

RUL, S. et al. **An Experimental Study on Performance Portability of OpenCL Kernels**, Dept. of Electronics and Information Systems, Ghent University, Gent, Belgica: 2010. Disponível em: < http://saahpc.ncsa.illinois.edu/papers/paper_2.pdf>. Acesso em: 25 de Julho de 2010.

RYOO, S. et al. **Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA**, Proceedings of the 13th ACM SIGPLAN Symposiun on Principles and Practice of Parallel Programming, Salt Lake City, UT, EUA: ACM, 2008, p. 73-82.

UENG S. et al. **CUDA-lite: Reducing GPU Programming Complexity**, Languages and Compilers for Parallel Computing: 21th International Workshop, Edmonton, Canada: Springer-Verlag: 2008, p. 1-15.

ZHIRNOV, VICTOR V. et al. **Limits to Binary Logic Switch Scaling – A Gendanken Model**, Proceedings of the IEEE, Vol. 91, No. 11, EUA: IEEE Press , 2003, p. 1934-1939.