FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO “EURÍPIDES DE MARÍLIA” – UNIVEM
CURSO DE CIÊNCIA DA COMPUTAÇÃO

JOSÉ HENRIQUE MADEIRA CIMINO

ANÁLISE DE PARALELISMO EM ARQUITETURA MULTICORE COM USO DE UNIDADE DE PROCESSAMENTO GRÁFICO

MARÍLIA
2011

JOSÉ HENRIQUE MADEIRA CIMINO

ANÁLISE DE PARALELISMO EM ARQUITETURA MULTICORE COM USO DE UNIDADE DE PROCESSAMENTO GRÁFICO

Trabalho de Curso apresentado ao Curso de Ciência da Computação da Fundação de Ensino "Eurípides Soares da Rocha", mantenedora do Centro Universitário Eurípides de Marília – UNIVEM, como requisito para obtenção do grau de Bacharel em Ciência da Computação.

Orientador:
Prof. Ms. MAURICIO DUARTE

MARÍLIA
2011

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL

José Henrique Madeira Cimino

ANÁLISE DE PARALELISMO EM ARQUITETURA MULTICORE COM USO DE GPU

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Ciência da Computação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Ciência da Computação.

Nota: 7.0 (sete)

Orientador: Mauricio Duarte ____________________

1º. Examinador: César Giacomini Penteado ____________________

2º. Examinador: Renata Aparecida de Carvalho Paschoal ____________________

Marília, 29 de novembro de 2011.

CIMINO, José Henrique Madeira. **Análise de Paralelismo em Arquitetura Multicore com Uso de Unidade de Processamento Gráfico.** 2011. 48 f. Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha de Marília, 2011.

RESUMO

O uso computacional da unidade de processamento gráfico (GPU) no auxílio a unidade central de processamento (CPU), para aplicações de propósitos gerais, e não apenas gráficos, pode ser de grande ajuda e pode haver aumento no desempenho da aplicação. Portanto para este projeto foi feito testes e análises com a programação CUDA, que permite este tipo de paralelismo entre a CPU e a GPU.

Palavras-chave: Unidade central de processamento, Unidade de processamento gráfico, paralelismo, cuda, desempenho.

CIMINO, José Henrique Madeira. **Análise de Paralelismo em Arquitetura Multicore com Uso de Unidade de Processamento Gráfico.** 2011. 48 f. Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha de Marília, 2011.

ABSTRACT

The use of computational GPU drive in helping the CPU for general purpose applications, not just graphics, can lead to an improvement in application performance. In this project were made tests and analysis with the CUDA programming, which allows this kind of parallelism between the CPU and GPU.

Keywords: cpu, gpu, parallelism, cuda, performance.

LISTA DE ILUSTRAÇÕES

Figura 1 - Arquitetura de Von Neumann (Wikipedia, 2011). 11
Figura 2 - Evolução dos processadores Intel (Cache Intel, 2011). 13
Figura 3 - Chip 4004 Intel (Museu do Computador, 2011). 14
Figura 4 - Chip 8088 Intel (IC UFF, 2011). 14
Figura 5 - Chip 286 Intel (Museu do Computador, 2011). 15
Figura 6 - Chip Pentium M (Tom's Hardware, 2011). 15
Figura 7 - Core Duo (Tec Mundo, 2011). 16
Figura 8 - Gargalo presente na arquitetura Single-core (Unicamp, 2011). 17
Figura 9 - Ausência do gargalo na arquitetura Multicore (Unicamp, 2011). 17
Figura 10 - CPU x GPU (Kerem, 2011). 18
Figura 11 - Detalhes do Device usado no projeto (Própria). 26
Figura 12 - Exemplo temporizador da API CUDA (NVIDIA, 2011). 27
Figura 13 - Configurando a Cópia zero Host (NVIDIA, 2011). 29
Figura 14 - Espaços de memória do Device (NVIDIA, 2011). 30
Figura 15 - Versão do driver CUDA instalado (Própria). 32
Figura 16 - CPU esperando resultados da GPU (Própria). 32

LISTA DE TABELAS

Tabela 1- Tempo de execução da ordenação de vetores com tamanhos diferentes. Linguagem C + CUDA (Própria). 34
Tabela 2 - Tempo de execução da ordenação de vetores com tamanhos diferentes. Linguagem C (Própria). 35

LISTA DE GRÁFICOS

Gráfico 1 - 5 elementos (Própria). 35
Gráfico 2 - 10 Elementos (Própria). 36
Gráfico 3 - 100 Elementos (Própria). 36
Gráfico 4 - 1.000 Elementos (Própria). 37
Gráfico 5 - 10.000 Elementos (Própria). 37
Gráfico 6 - 100.000 Elementos (Própria). 38
Gráfico 7 - 1.000.000 Elementos (Própria). 38

SUMÁRIO

INTRODUÇÃO 10

CAPÍTULO 1 – PROCESSADORES E GPUS 11
1.1. Histórico dos Processadores Intel 11
1.2. Core 2 Duo 16
1.3. Unidade de Processamento Gráfico (GPU) .... **Erro! Indicador não definido.**

CAPÍTULO 2 – Computação Paralela 19
2.1. Diferenças entre Sistemas Paralelos e Sistemas Distribuídos 21
2.2. Práticas de Programação Paralela 21
2.2.1. Decomposição do Problema 22

CAPITULO 3 – CUDA 23
3.1. Diferenças entre Host e Device 23
3.2. Benefício máximo de desempenho 25
3.3. Ambiente de programação 26
3.4. Métricas de Desempenho 26
3.4.1. Tempo 27
3.5. Otimização de Memória 28
3.5.1. Transferência de dados entre Host e Device 28
3.5.2. Cópia Zero 28
3.5.3. Espaços de memória do Device 29

CAPITULO 4 – DESCRIÇÃO DO PROJETO 31
4.1. Download e Instalação CUDA 31
4.2. Compilando CUDA 32
4.3. Algoritmos e Métodos 33

CAPITULO 5 – ANÁLISES, TESTES E RESULTADOS 34

CAPITULO 6 – CONCLUSÃO 40

REFERÊNCIAS 41

ANEXO 46

INTRODUÇÃO

A GPU em auxílio para aceleração de aplicações de propósito geral pode ser uma grande solução para quem quer maior desempenho e não apenas em aplicações gráficas como jogos, filme, processamentos de imagens e outros. A primeira sessão aborda um pouco sobre a história dos processadores da família Intel e sua rápida evolução na procura de maior desempenho e uma pequena abordagem sobre o que é e o funcionamento da placa GPU.

No auxilio da GPU em aplicações de propósitos gerais é necessário o uso de paralelismo, e é de grande importância entender seu funcionamento para uma boa prática de programação paralela. Para haver esse paralelismo controlado pelo programador é necessário o uso de uma linguagem de programação chamada CUDA, desenvolvida pela empresa NVIDA. CUDA é uma extensão da linguagem de programação C, permitindo o paralelismo entre a CPU e a GPU.

Este projeto tem como objetivo analisar o desempenho de paralelismo entre o processador (CPU) e a placa de vídeo (GPU) em aplicações de propósitos gerais. Para isso foi usado um notebook com processador desenvolvido pela Intel. O modelo é o Core 2 Duo, composto por dois núcleos. Foi usado também uma placa de vídeo do modelo GeForce 8600 M GT da marca NVIDIA. É essencial o uso da placa de vídeo da marca NVIDIA para esse tipo de paralelismo, pois é uma tecnologia desenvolvida pela empresa NVIDIA.

A biblioteca para a implementação desse tipo de paralelismo é chamada CUDA, uma extensão da linguagem de programação C. CUDA permite o programador dividir as tarefas de um programa entre a CPU e a GPU com finalidade de aumentar o desempenho.

CAPÍTULO 1 – PROCESSADORES E GPUS

1.1. Histórico dos Processadores Intel

O conceito de processador surgiu em 1970, antes disso (anos 40, 50 e 60) os processos eram processados de forma primitiva comparada com as de hoje. Eram válvulas e transistores em máquinas enormes para que houvesse um processamento com sucesso, porem lento.

O matemático John Von Neumann e sua equipe desenvolveram o primeiro conceito de computador que tinha a capacidade de armazenar seus programas em um mesmo espaço de memória que seus dados, assim facilitando sua manipulação. Ainda hoje, a maior parte dos computadores é concebida tendo, como idéia fundamental, os princípios desta arquitetura. Suas principais características são:

- Possuir uma unidade de processamento central, para a execução de operações lógicas e aritméticas;
- Possuir uma unidade de controle, responsável por determinar o seqüenciamento das instruções a serem executadas por meio de sinais de controle;
- Instruções dos programas armazenadas de maneira seqüencial, facilitando a busca;
- Existência de registradores dedicados ao armazenamento dos operandos e dos resultados das operações;
- Unidade de armazenamento central, na qual são guardados programas e dados, de forma compartilhada;
- Existência de um único barramento do sistema, o qual deve ser usado de forma compartilhada para a transferência de dados e instruções entre os diversos blocos.

A Figura 1 mostra o modelo de arquitetura de computadores de Von Neumann:

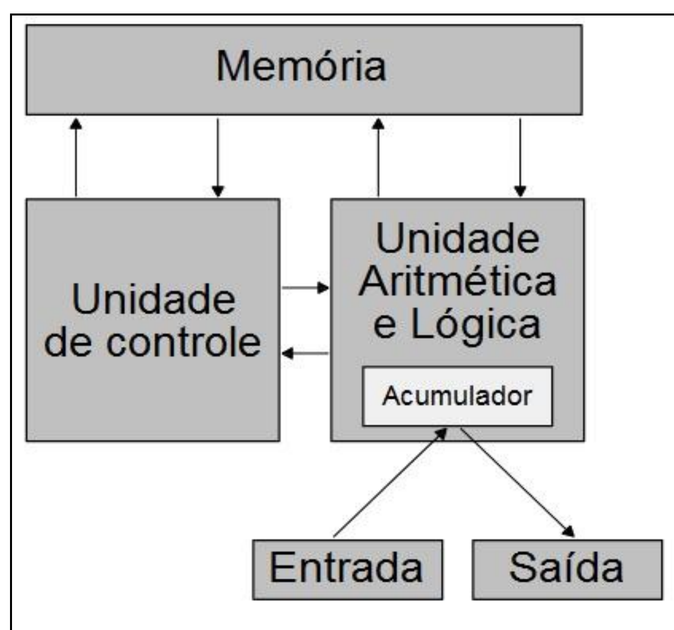

Figura 1 - Arquitetura de Von Neumann (Wikipedia, 2011).

A arquitetura foi largamente difundida, influenciando muitos projetos subseqüentes de outras máquinas.

No ano de 1960, a IBM lançou o primeiro computador (IBM 7030 Stretch) totalmente processado com transistores, no lugar de válvulas. Em 1964 os processadores passaram a usar o conceito de "circuito integrado", interligando várias operações em um mesmo circuito, graças ao IBM 360.

Inícios da década de 70 surgiram as CPU's desenvolvidas totalmente em circuito integrado em um único *chip* de silício, unidade única central.

O processador é a unidade central de processamento de computador ou sistema computacional. É um circuito integrado que executa instruções de máquina (*boolean*), realiza cálculos e é responsável por controlar todos os outros *chips* do computador.

Há dois componentes mais importantes em um sistema: o módulo que interpreta as instruções e o módulo que executa as funções lógicas e aritméticas.

"Processador: Seus principais componentes são a unidade de controle, os registradores, a unidade lógica e aritmética (ULA) e a unidade de execução de instruções." (Stallings, 2003).

É pequeno, formado por milhões de transistores e considerado o cérebro do computador.

Este projeto é voltado para processadores família Intel, onde serão realizados testes de paralelismo entre a CPU e a GPU. A Figura 2 mostra a linha do tempo de uns dos mais importantes processadores desenvolvidos pela Intel:

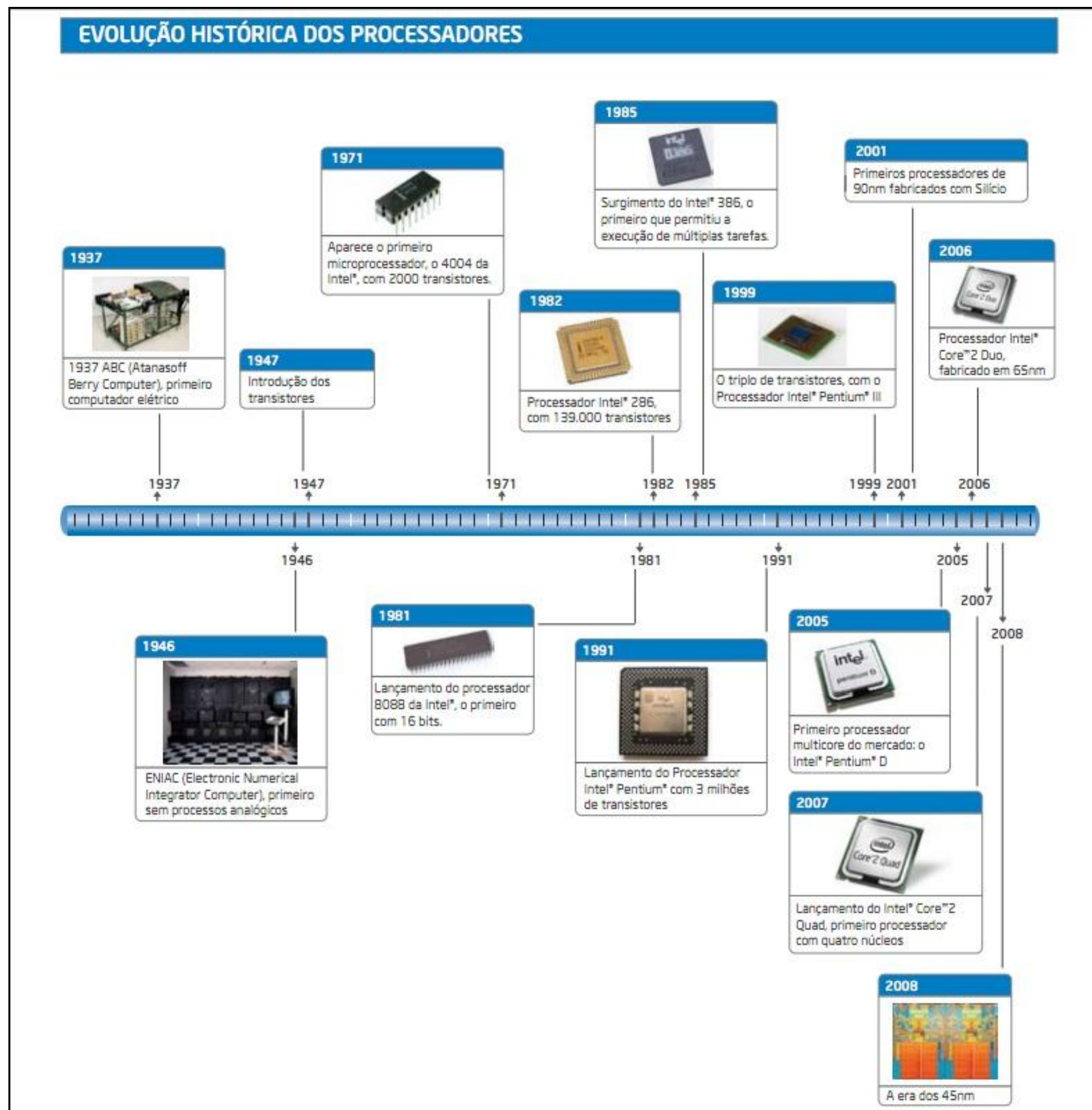

Figura 2 - Evolução dos processadores Intel (Cache Intel, 2011).

Em 1969 o *chip* 4004 junto com memória ROM, memória RAM e uma porta de I/O (entrada e saída) foram desenvolvidos pela Intel para uma empresa chamada *Nippon Cálculo Machine Corporation* para a construção da calculadora *Busicom* 141-PF.

Esse *chip* 4004 é constituído de cerca de 2300 transistores, 4-*bits* paralelos, CPU com 46 instruções, métodos aritméticos em binário e decimal, 10.8 ciclos de instruções por microssegundos, diretamente compatível com ROMs MCS-40 podendo acionar diretamente até 32.768 *bits* de memória e as RAMs podendo acionar diretamente até 5120 *bits* de memória. Velocidade de 108KHz. A figura 3 mostra o *chip* 4004.

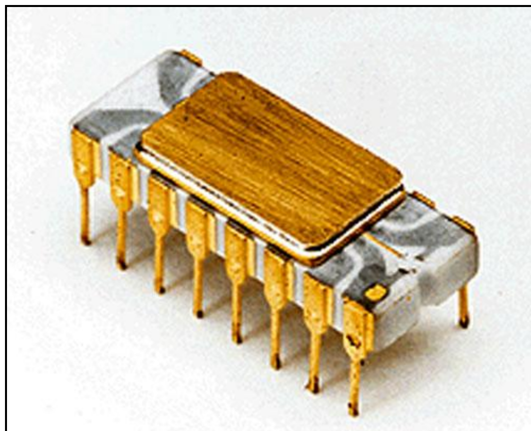

Figura 3 - Chip 4004 Intel (Museu do Computador, 2011).

De 1979 a 1981 foi desenvolvido o primeiro processador 16 *bits*, Intel 8088. Apesar de ser 16 *bits*, seu barramento de dados era de 8 *bits*. Considerado o novo cérebro *hit* da IBM, foi o sucesso da Intel na época. Composto por 29.000 transistores e atingia uma velocidade de 5MHz a 10MHz. Acesso de até 1MB de memória. A figura 4 mostra o *chip* 8088:

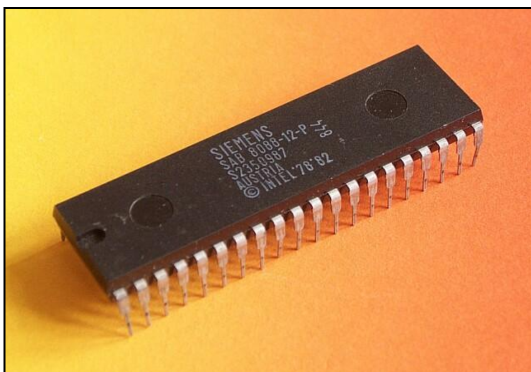

Figura 4 - Chip 8088 Intel (IC UFF, 2011).

O processador Intel 286 desenvolvido em 1982, verdadeiramente 16 *bits* ele tinha a capacidade de acessar 16MB de memória RAM. Tem a capacidade de executar vários programas "ao mesmo tempo", múltiplas tarefas, a partir dos sistemas operacionais *Windows*, por exemplo. Composto por 134.000 transistores e tem uma velocidade de 6MHz e mais tarde em outras edições chegou a 20MHz. A figura 5 abaixo mostra a mudança no formato no novo *chip* Intel 286:

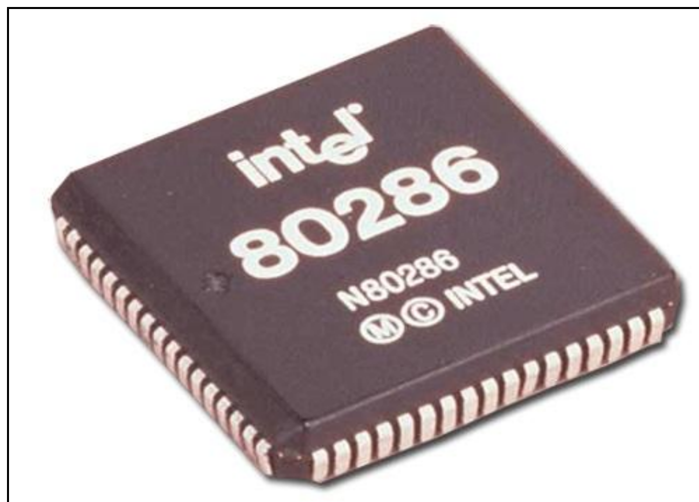

Figura 5 - Chip 286 Intel (Museu do Computador, 2011).

A arquitetura *Pentium* M é uma evolução do *Pentium* II, *Pentium* III, mas é diferente da arquitetura *Pentium* 4. Quando usado em notebooks integrado com *chipset* Intel 855 ou 915 e rede *wireless* Intel Pro, é chamado de Centrino.

Externamente o *Pentium* M trabalha igual o *Pentium* 4 transferindo quatro dados por pulso de *clock*, fazendo com que o barramento local tenha um desempenho quatro vezes maior do que seu *clock* atual. Esta técnica é chamada de *Quad Data Rate*. O Pentium M manteve o formato do *chip* 286 como mostra a figura 6:

Figura 6 - Chip Pentium M (Tom's Hardware, 2011).

Conforme a tecnologia dos processadores foi crescendo, seus transistores foram diminuindo de tal forma que a partir do Pentium 4 eles já estavam tão pequenos e numerosos que ficou muito difícil de aumentar o *clock* devido a limitações físicas da época e ao superaquecimento.

A melhor solução foi adicionar mais um núcleo na arquitetura, não alterando o tamanho do *chip*

como mostra a figura abaixo, utilizando a tecnologia *multicore*. Cada núcleo não trabalha em uma freqüência tão alta, e mesmo assim quase chega ao dobro de *clock*. Por exemplo, um *single-core* 4.2 GHz pode ser atingido por um *dual-core* 2.1 GHz, não se pode dizer efetivamente que são equivalentes devido ao tempo de escalonamento que se perde, mas é uma solução muito boa aumentando o desempenho e mantendo o tamanho do *chip*, como mostra a figura 7:

Figura 7 - Core Duo (Tec Mundo, 2011).

1.2. Core 2 Duo

A linha de processadores *Core 2 Duo*, da Intel, veio para substituir o *Pentium* que foi um grande sucesso, e para que fosse aceito como o melhor substituto ele teria que apresentar alto desempenho.

O *Core 2 Duo* foi desenvolvido para computadores de mesa e para *notebooks*, o que não acontecia desde 2003, quando houve a divisão entre as linhas Pentium 4 e Pentium M. O *Core 2 Duo* foi desenvolvido com dois núcleos de processamento, a Intel decidiu investir na melhoria do barramento e da memória *Cache*, assim permitindo o aumento de desempenho sem o grande aumento de frequência, consumo de energia e do calor dissipado, que eram as armas usadas no Pentium, um *clock* alto, porem muito consumo de energia e muito calor liberado. Ainda sim o *Core 2 Duo* apresentou bons resultados nos testes.

O conceito de multitarefa usado nos sistemas operacionais só passou a ser um fato real a partir do lançamento do processador com dois núcleos ou mais. Mesmo com processadores de um único núcleo a sensação de estar executando vários programas de uma única vez era tão real que se usava o termo multitarefa, porem não era em paralelo. Cada programa tinha um tempo no processador, dado pelo escalonador, e então havia trocas para ver quem seria o próximo a ser processado. Já com dois ou mais núcleos, pode-se dizer que os programas estão sendo executados ao mesmo tempo e com isso, o aumento de desempenho é real.

O grande problema de um único core era aumentar sua frequência de processamento em um

pequeno espaço físico, pois esquentaria muito, podendo levar a destruição do núcleo. E mesmo que isso fosse possível, não seria algo fácil e rápido de se desenvolver. O uso de mais núcleos de processamento não tem esse problema de aquecimento, pois a frequência de cada núcleo esta em seu estado ideal para processar em alta velocidade e não esquentar. Segundo Cardoso *et al* (2005), outra limitação do único núcleo é a estreita banda de dados, aliada a grande diferença entre a velocidade do processador e a da memória, faz com que 75% do tempo da CPU, em média, seja gasto esperando por resultados dos acessos à memória. A figura 8 mostra o gargalo que ocorre em um único *core*.

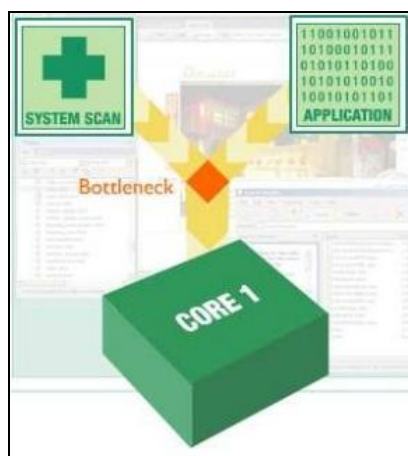

Figura 8 - Gargalo presente na arquitetura Single-core (Cardoso *et al*., 2011).

Na arquitetura *multicore* o sistema operacional trata cada núcleo como um processador diferente, dividindo as tarefas e tornando o conceito de multitarefa real, como mostra a figura 9.

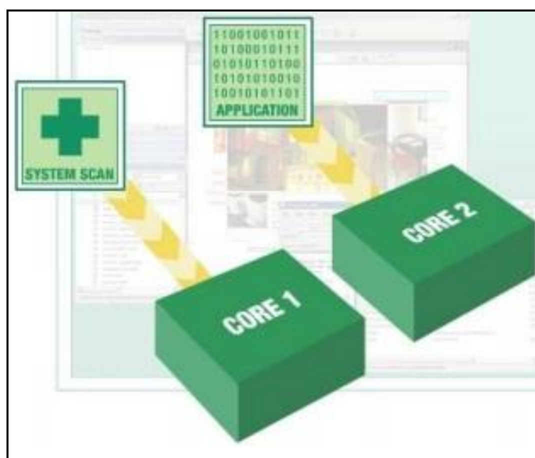

Figura 9 - Ausência do gargalo na arquitetura Multicore (Cardoso *et al.*, 2011).

Este problema de arquitetura está resolvido, mas outro problema aparece, a grande quantidade de softwares sequenciais. Se cada core for responsável por um *software* diferente fica claro o grande desempenho de processamento paralelo, porém se um único software estiver em execução, por exemplo, pelo *core* 1, o *core* 2 ficaria ocioso e não estaria aproveitando 100% do desempenho da arquitetura *multicore*. Porém se esse único *software* em execução não fosse um código sequencial, como descrito por Von Neumann (1945) em seu relatório embrionário, e sim fossem desenvolvidos para arquiteturas paralelas

ou sistemas distribuídos, o *core* 2 não estaria mais ocioso e sim ajudando na execução desse *software*, assim tendo 100% de aproveitamento dessa nova arquitetura e seu grande desempenho.

O grande desafio é desenvolver esses *softwares* em paralelo, pois não é uma tarefa fácil, mas estudos estão crescendo cada vez este assunto.

1.3. Unidade de Processamento Gráfico (GPU)

Para usuários que jogam games para computador, assistem filmes, customizam imagens, e muito mais, uma boa Unidade de Processamento Gráfico é indispensável. As duas maiores empresas de desenvolvimento de Unidade de Processamento Gráfico é Nvidia e a ATI.

Mesmo com o avanço da tecnologia e aumento de velocidade dos processadores, as exigências crescem muito mais rápido que sua evolução, assim exigindo mais de computadores em paralelo, destinados a processar a mesma informação, dividindo tarefas para haver uma resposta mais rápida. Portanto o uso de paralelismo está crescendo cada vez mais.

A CPU é capaz de realizar cálculos matemáticos complexos com altas velocidades, porém, a GPU também é capaz de realizar matemáticos complexos avançados de formar bem eficaz. A GPU em auxílio para aceleração de aplicações de propósito geral pode ser uma grande solução para quem quer maior desempenho. A figura 10 mostra a diferença da divisão de componentes entre a CPU e a GPU.

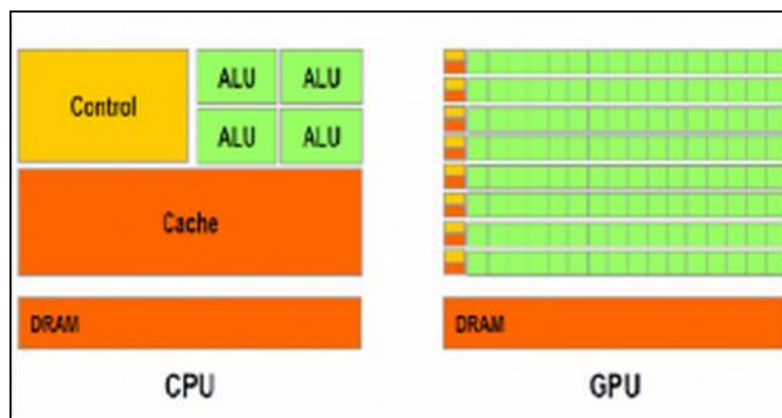

Figura 10 - CPU x GPU (Kerem, 2011).

A Figura 10 mostra a diferença entre a CPU e a GPU, o modo como são distribuídas as unidades lógicas aritméticas (ALU).

CAPÍTULO 2 – Computação Paralela

Computação paralela é uma forma de computação onde vários cálculos são realizados simultaneamente, podendo transformar um problema enorme em vários problemas menores e assim serem resolvidos corretamente em paralelo.

O uso de paralelismo cresceu muito há algum tempo, principalmente na computação de alto desempenho devido a limitações físicas e ao aquecimento dos processadores de um único núcleo na tentativa de aumentar sua frequência de processamento. Essa técnica de paralelismo deu origem a arquitetura de computadores com dois ou mais núcleos em uma única máquina, diminuindo suas frequências e não dependendo mais da limitação física de cada um e mesmo assim ganhando um alto desempenho.

Não existe apenas o paralelismo de processadores multicore em apenas uma máquina, há outros tipos como um cluster, que é um aglomerado de computadores trabalhando em paralelo a partir de sistemas distribuídos como se fosse uma única máquina.

"Um sistema Distribuído é um conjunto de elementos que se comunicam através de uma rede de interconexão e que utilizam software de sistema distribuído. Cada elemento é composto por um ou mais processadores e uma ou mais memórias." (Pitanga, 2004)

"Um sistema Distribuído também pode ser classificado como um conjunto de computadores com seu próprio clock e que não possuem memória compartilhada." (Pitanga, 2004)

"Um sistema distribuído é um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente." (Tanenbaum, 2007).

Sistemas Distribuídos tem suas vantagens e desvantagens, como todo sistema. Se comparado com sistemas tradicionais em execução de algumas tarefas, o sistema distribuído tem suas vantagens como custo e desempenho, porém seu desenvolvimento é complexo e pode haver falhas na segurança.

A programação paralela é mais difícil de implementar que programas sequenciais, pois depende muito da comunicação e a sincronização dos dispositivos em paralelo. O aumento de velocidade por resultado de paralelismo é dado pela lei de Amdahl.

Os principais objetivos da programação paralela são: reduzir o tempo de processamento (*wall clock time*); o custo de vários computadores trabalhando em paralelo pode ser menor que o custo de um super computador; recursos locais versus recursos não locais e restrições de memória.

Todavia, um sistema paralelo pode não ter utilidade se não tiver um programa paralelo que seja capaz de dividir suas tarefas, que sincronize seus resultados dado ao trabalho atribuído a cada elemento de processamento e compartilhe a memória de forma eficiente.

Diferentes arquiteturas foram propostas para se alcançar paralelismo, havendo diferenças na interligação entre elas. As mais conhecidas são a de taxonomia de Flynn (Rocha, 2008) e a de Duncan (Duncan, 1990).

A taxonomia de Flynn está classificada em quatro classes de arquitetura de computadores, elas são:

SISD (*Single Instruction Single Data*): Único fluxo de instruções e um único conjunto de dados. Tipo de arquitetura mais simples, devido a um único dado opera a cada instrução. Um exemplo de SISD é a arquitetura de Von Neumann.

SIMD (*Single Instruction Multiple Data*): Único fluxo de instrução e múltiplos conjuntos de dados. Uma única instrução é aplicada a múltiplos dados, assim produzindo mais resultados. Muito aplicado em matrizes e vetores.

MISD (*Multiple Instruction Single Data*): Múltiplo fluxo de instruções e um único conjunto de dados. Operações diferentes são aplicadas a um único conjunto de dados.

MIMD (Multiple Instruction Multiple Data): Múltiplo fluxo de instruções e múltiplos dados. CPUs diferentes executam programas diferentes, compartilham a mesma memória através de barramentos lógicos.

Duncan classificou duas formas de arquitetura paralela:

Arquitetura Síncrona: Um relógio global é responsável por manipular todas as operações, esse grupo é formado pela arquitetura SIMD da taxonomia de Flynn.

Arquitetura Assíncrona: Os processadores são autônomos, não possuem um controlador de operações, e são compostos basicamente pelas arquiteturas MIMD da taxonomia de Flynn.

Outra forma de classificar o paralelismo é em Paralelismo Implícito e Explícito.

O Paralelismo Implícito diz respeito ao compilador controlar, sincronizar, detectar e atribuir tarefas as execuções dos programas. As vantagens são: não cabe ao programador os detalhes do paralelismo, é muito eficaz em soluções gerais e é bem flexível. Sua desvantagem é não conseguir uma solução eficiente para todos os casos.

O Paralelismo Explícito está totalmente voltado ao programador, ele é responsável por determinar as tarefas de execução em paralelo, controlar essas tarefas, conhecer a fundo a arquitetura em que esta trabalhando para conseguir o máximo desempenho do paralelismo. Sua vantagem está ligada a programadores experientes, pois conseguem soluções muito eficientes para problemas específicos. Suas desvantagens são: difícil de implementar, grande quantidade de detalhes para se preocupar e pouca portabilidade entre maquinas de diferentes arquiteturas.

A programação paralela tornou-se possível a partir do desenvolvimento de sistemas operacionais multitarefa, *multithread* e paralelos. Tem como característica a execução simultânea da várias partes de uma mesma aplicação.

A programação distribuída tornou-se possível a partir da popularização das redes de computadores. Tem como característica a execução de aplicações cooperantes em máquinas diferentes, um exemplo de programação distribuída é o *cluster*.

O *cluster* é uma arquitetura composta por vários computadores interligados por uma rede, onde se

comunicam.

2.1. Diferenças entre Sistemas Paralelos e Sistemas Distribuídos

Sistemas paralelos são fortemente acoplados, compartilham *hardware* ou se comunicam através de um barramento de alta velocidade. Sistemas distribuídos são fracamente acoplados.

O comportamento de um Sistema paralelo, ao considerar o tempo, é mais previsível, já os Sistemas distribuídos são mais imprevisíveis devido ao uso da rede (*hub*, *switch*, roteadores, *firewall*, etc.) e a falhas.

O Sistema distribuído é bastante influenciado pelo tempo de comunicação pela rede, já o Sistema paralelo desconsidera o tempo de troca de mensagens.

Em geral Sistemas paralelos tem o controle de todos os recursos computacionais, já o Sistema distribuído tende a empregar recursos de terceiros.

-Vantagens:

Sistemas Paralelos e Distribuídos: usam melhor o poder de processamento; apresentam um melhor desempenho; permitem compartilhar dados, recursos e reutilizar serviços já disponíveis, assim como apresentar maior confiabilidade e atender um maior número de usuários.

-Dificuldades:

Sistemas Paralelos e Distribuídos: Desenvolver, gerenciar, e manter o sistema; controlar o acesso corretamente a dados e a recursos compartilhados; evitar que falhas de máquinas ou da rede comprometam o funcionamento do sistema; garantir a segurança do sistema e o sigilo dos dados trocados entre máquinas; lidar com a heterogeneidade do ambiente; etc.

Os testes realizados usando o paralelismo entre CPU e GPU utiliza a programação paralela, pois utiliza o mesmo computador compartilhando *hardwares* fortemente acoplados por um barramento de alta velocidade, a CPU e a GPU.

2.2. Práticas de Programação Paralela

O processo de decompor um problema em partes menores e bem definidas não é uma tarefa fácil. Um programador com fortes habilidades de pensamento computacional não apenas analisa, mas também transforma a estrutura de um problema, determinando quais partes são inerentemente seriais, quais partes são receptivas à execução paralela de alto desempenho, e as alternativas envolvidas na mudança de partes da primeira categoria para a segunda. Uma combinação forte de conhecimento do domínio e habilidades de pensamento computacional normalmente é necessária para criar soluções computacionais bem-sucedidas para problemas de domínio desafiadores.

2.2.1. Decomposição do Problema

O conceito de paralelizar grandes problemas computacionais normalmente é simples, mas na prática se torna desafiador. A chave é identificar o trabalho a ser realizado por cada unidade de execução paralela de modo que o paralelismo seja bem aproveitado.

Quando o conjunto de dados é grande o suficiente e o cálculo mais exigente tiver sido paralelizado, pode-se efetivamente paralelizar o cálculo menos exigente. Como alternativa pode-se tentar executar simultaneamente vários *kernels* pequenos, cada um correspondendo a uma tarefa. Uma técnica alternativa para reduzir o efeito das tarefas sequenciais é explorar o paralelismo de dados de uma maneira hierárquica. Em uma implementação típica de *Message Passing Interface* (MPI), uma aplicação normalmente distribuiria grandes pedaços associados a nós de um *cluster* de computador em rede. Usando o *host* de cada nó para calcular pedaços menores da aplicação, poderia alcançar um ganho de velocidade. Cada nó pode usar um *device* CUDA para ajudar no cálculo ganhando mais velocidade. MPI e CUDA podem ser usados de uma maneira complementar nas aplicações para alcançar um nível mais alto de velocidade com grandes *datasets*.

CAPITULO 3 – CUDA

"CUDA é uma plataforma de computação paralela e um modelo de programação desenvolvido pela NVIDIA. Ela permite aumentos significativos de desempenho computacional ao aproveitar a potência da unidade de processamento gráfico (GPU)." (NVIDIA Corporation, 2011)

As GPUs são dispositivos paralelos com alta capacidade de executar cálculos aritméticos mais complexos, usada muito em jogos, imagens, filmes e outros, onde exige um processamento maior. As GPUs, por terem memória própria, são perfeitas para esse tipo de processamento.

O principal intuito da CUDA é facilitar a programação de aplicativos usando o poder da GPU, o que consiste em programação paralela. Mas isso só é possível com GPUs da marca NVIDIA.

A programação CUDA, por ser paralela, envolve o código em execução em duas plataformas diferentes: no *host* com um ou mais CPUs e o *device* com uma ou mais GPUs NVIDIA CUDA. Enquanto os *devices* NVIDIA são muito associados com renderização de gráficos, também são utilizados em cálculos aritméticos capaz de executar milhares de *threads* em paralelo. Essa capacidade os torna adequados para cálculos que pode melhorar o processamento paralelo. No entanto, o *device* é baseado em um design bem diferente do *host*, e é importante compreender essas diferenças e como elas determinam o desempenho da aplicação CUDA e como usar CUDA de forma eficaz.

As próximas seções foram escritas através de estudos feitos do guia da NVIDIA para programação CUDA (CUDA C Best Practices Guide, 2011).

3.1. Diferenças entre Host e Device

As principais diferenças estão no acesso a *threads* e a memória:

-Recurso das *Threads*: A execução de *pipelines* em um *host* pode aguentar um número limitado de *threads* ao mesmo tempo. Mesmo servidores que têm vários núcleos de processamento (*core*), podem executar poucas *threads* ao mesmo tempo, por comparação, a unidade mais simples de execução paralela em um dispositivo CUDA pode várias *threads*. Todas as GPUs da NVIDIA podem suportar, pelo menos, 768 *threads* ativos simultaneamente por multiprocessador, e algumas até 1024 ou mais. Em dispositivos com 30 multiprocessadores leva a cerca de 30.000 *threads* ativas[1].

-*Threads*: *Threads* em CPU são normalmente entidades pesadas. O sistema operacional deve fazer trocas de *threads* dentro e fora dos canais de execução da CPU, para fornecer capacidade *multithreading*. Trocas de contexto (quando dois segmentos são trocados) são caras e lentas.

Em comparação com as da CPU, *threads* em GPUs são extremamente leves. Em um sistema típico, milhares de *threads* estão na fila para execução. Se a GPU deve esperar por um resultado de uma *thread* ela

[1] Informação obtida através do manual CUDA C Best Practices Guide

simplesmente começa a executar outras que não estão esperando por este resultado. Isso é possível pelo fato de que registros separados são alocados para todas as *threads* ativas, sem que a troca de registros ou estado precise ocorrer entre as *threads* da GPU. Os recursos ficam alocados para cada *thread*, até que as *threads* terminem de executar e então os recursos são desalocados.

-Memória (RAM): Tanto o *host* quanto o *device* tem memória principal (RAM). No *host* a RAM é geralmente acessada por igual a todos os códigos (dentro das limitações impostas pelo sistema operacional). No *device*, a memória RAM é dividida virtualmente e fisicamente em diferentes tipos, cada um contendo seu propósito especial e atendendo diferentes necessidades.

Com isso, conclui-se que as *threads* no *device* (GPU) são bem mais aproveitadas e de uma forma mais eficaz, enquanto na CPU as *threads* são pesadas e não são aproveitadas de uma forma eficaz. Percebe-se também que o *device* por ter sua própria memória facilita na execução de processos e de suas *threads*, aumentando assim seu desempenho.

Estas são as principais diferenças de *hardware* entre *host* (CPU) e *device* (GPU) com respeito à programação paralela.

Devido às diferenças entre *host* e *device*, é importante para a aplicação que cada sistema de *hardware* esteja fazendo o trabalho que faz melhor. As seguintes questões devem ser consideradas ao determinar quais partes de um aplicativo, ao ser particionado, será executado no *device*:

- O *device* é ideal para executar cálculos com vários dados ao mesmo tempo em paralelo. Isso geralmente envolve aritmética sobre grande conjunto de dados como matrizes, onde a mesma operação pode ser realizada através de milhares de elementos, ao mesmo tempo. Este é um requisito para o bom desempenho em CUDA: o *software* deve usar um grande número de *threads*. O apoio para a execução de inúmeras *threads* em paralelo deriva do uso da arquitetura CUDA e o uso de seu modelo de *threading* que é muito leve.

- Deve haver alguma coerência no acesso à memória do *device*. Certos padrões de acesso à memória permitem o hardware a unir grupos de leituras e/ou gravações da vários itens de uma mesma operação. Dados que não podem ser dispostos a essa união, não são usados de forma eficiente no CUDA.

A transferência de dados entre *host* e *device* no CUDA acontece através do barramento. Estas transferências, em termos de desempenho, são caras e devem ser minimizadas, porém são rápidas. Este custo tem várias ramificações:

- A complexidade de operações deve justificar o custo de transferir dados do *device* e para o *device*. Código que transfere dados para o uso breve e para o uso de poucas *threads*, não haverá beneficio no ganho de desempenho, pelo contrário, perderá muito tempo transferindo e executando pouco. O código ideal é aquele em que muitas *threads* executam uma quantidade substancial de dados. Por exemplo, a transferência de duas matrizes para o *device* para executar uma adição de matriz e, em seguida, transferir o resultado de volta para o *host*, mesmo ganhando desempenho não será muito notado. A questão aqui é o numero de

operações realizadas por elemento de dados transferido.

- Dados devem ser mantidos no *device* o maior tempo possível, pois as transferências devem ser mínimas. Programas que executam vários *kernels* no mesmo dado devem favorecer, deixando os dados no *device* entre as chamadas de *kernel*, ao invés de transferir resultados intermediários para o *host* e, em seguida, enviá-los de volta ao *device* para próximos cálculos.

O *device* é ótimo para executar cálculos complexos e devido a suas *threads* e sua memória principal também é ótimo para grandes quantidades de dados. Porém em CUDA quando há o paralelismo entre *host* e *device*, há uma grande troca de dados entre eles, e pode haver congestionamento no barramento, assim o desempenho que seria ótimo, passa a cair. Por isso o programador deve saber a hora de fazer essa troca de informações entre o *host* e o *device*, para que haja o ganho de desempenho. Caso partes do código não possa ser paralelizado, então se entende que o melhor é deixar o *host* processar, pois seria mais rápido do que fazer essa troca excessiva de informação.

3.2. Benefício máximo de desempenho

Para obter o benefício máximo do CUDA, primeiro foco é encontrar maneiras de paralelizar o código sequencial. A quantidade de benefícios de desempenho que será percebido de um aplicativo executado em CUDA depende inteiramente do grau em que ele pode ser paralelizado. Como mencionado anteriormente, códigos que não podem ser suficientemente paralelizados devem ser executados no *host*, a menos que isso resultaria em transferências excessivas entre *host* e o *device*.

A lei de Amdahl especifica a velocidade máxima que pode ser esperado pela paralelização de um programa seqüencial. Ele afirma que a velocidade máxima de (*Gaonho*) de um programa é:

Ganho = 1 / ((1-Fm) + Fm/Ge)

Onde *Fm* é a fração de tempo total de execução de séries tomadas por uma parte do código que pode ser paralelizado e *Ge* é o número de processadores que fazem parte da paralelização do código em execução. Na maioria dos casos o ponto chave é: maior o *Fm*, maior o *Ganho*. Uma observação adicional está implícita nesta equação, quando o *Fm* é um número pequeno, há a possibilidade de aumentar *Ge* para melhorar o desempenho. Mas para obter um ótimo desempenho, melhores práticas sugerem tentar aumentar *Fm*, ou seja, aumentando a quantidade de código que pode ser paralelizado.

Essa fórmula criada por Amdahl mostra exatamente a influência que a paralelização de códigos tem para com o desempenho. Quanto mais porções do código podem ser paralelizadas, maior o desempenho do paralelismo. É claro também que a quantidade de processadores envolvidos afeta no resultado, de acordo com a fórmula de Amdahl, quanto mais processadores, maior o desempenho, mas isso não se aplica na realidade.

3.3. Ambiente de programação

Com cada geração de processadores NVIDIA, novos recursos são adicionados à GPU CUDA. Portanto é importante compreender as características da arquitetura.

Neste projeto será usada para análise o *device* GeForce 8600M GT da NVIDIA. A figura 11 mostra suas características CUDA:

```
Terminal — deviceQuery — 154×48
Last login: Sat Nov 12 12:49:09 on ttys001
MacBook-Pro-Ze-Cimino:~ marcosaquino$ /Developer/GPU\ Computing/C/bin/darwin/release/deviceQuery ; exit;
[deviceQuery] starting...
/Developer/GPU Computing/C/bin/darwin/release/deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Found 1 CUDA Capable device(s)

Device 0: "GeForce 8600M GT"
  CUDA Driver Version / Runtime Version          4.0 / 4.0
  CUDA Capability Major/Minor version number:    1.1
  Total amount of global memory:                 512 MBytes (536674304 bytes)
  ( 4) Multiprocessors x ( 8) CUDA Cores/MP:     32 CUDA Cores
  GPU Clock Speed:                               1.04 GHz
  Memory Clock rate:                             650.00 Mhz
  Memory Bus Width:                              128-bit
  Max Texture Dimension Size (x,y,z)             1D=(8192), 2D=(65536,32768), 3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers        1D=(8192) x 512, 2D=(8192,8192) x 512
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 8192
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             256 bytes
  Concurrent copy and execution:                 Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Concurrent kernel execution:                   No
  Alignment requirement for Surfaces:            Yes
  Device has ECC support enabled:                No
  Device is using TCC driver mode:               No
  Device supports Unified Addressing (UVA):      No
  Device PCI Bus ID / PCI location ID:           1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 4.0, CUDA Runtime Version = 4.0, NumDevs = 1, Device = GeForce 8600M GT
[deviceQuery] test results...
PASSED

Press ENTER to exit...
```

Figura 11 - Detalhes do Device usado no projeto (Própria).

É de grande importância o programador conhecer a arquitetura em que está desenvolvendo. Os programadores devem estar cientes de dois números de versão. O primeiro é a *compute capability*, e o segundo é o número da versão *Runtime* e a versão do *driver* CUDA. Neste caso o *Runtime* e o *drive* CUDA são respectivamente 4.0 e 4.0 e sua *compute capability* é 1.1.

A partir desses dados, o programador tem conhecimento da arquitetura e suas características para um bom desenvolvimento.

3.4. Métricas de Desempenho

Ao tentar otimizar o código CUDA, que se encontra no anexo A, é necessário saber como medir o desempenho com precisão e compreender o papel que a banda causa na medição do desempenho. Esta seção discute como medir corretamente o desempenho usando temporizador da CPU e eventos CUDA.

3.4.1. Tempo

A chamada CUDA e a execução *kernel* podem ser programadas usando temporizadores da CPU ou GPU. Esta seção examina a funcionalidade, vantagens e desvantagens de ambas as abordagens.

-Usando Temporizador CPU: Qualquer temporizador CPU pode ser usado para medir o tempo decorrido de uma chamada CUDA ou execução *kernel*. Ao usar o temporizador CPU, é fundamental lembrar que muitas funções da API da CUDA são assíncronas, isto é, eles retornam o controle das *threads* da CPU chamando antes de terminarem o seu trabalho. Todos os lançamentos do *kernel* são assíncronos. Portanto, para medir com precisão o tempo decorrido para uma chamada específica ou uma sequência de chamadas CUDA, é necessário sincronizar as *threads* da CPU com a GPU, chamando *cudaThreadSynchronize()* imediatamente antes de iniciar e finalizar o temporizador CPU.

A função *cudaThreadSynchronize()* bloqueia as chamadas da *thread* da CPU até que todas a chamadas CUDA antes emitidas pela *thread* sejam terminadas. Embora também seja possível sincronizar a *thread* da CPU com um fluxo ou evento sobre a GPU, estas funções de sincronização não são adequadas para o código de tempo.

A função *cudaStreamSynchronize()* bloqueia a *thread* da CPU até que todas as chamadas CUDA previamente emitidas para o determinado fluxo forem terminadas.

A função *cudaEventSynchronize()* bloqueia um determinado evento até que um fluxo particular tenha sido registrado pela GPU.

-Usando Temporizador GPU CUDA: O evento da API da CUDA oferece chamadas que criam e destroem eventos, gravam eventos e converte diferenças de *timestamp* em valores de ponte flutuante em milissegundos. A figura 12 mostra o seu uso em código:

```
cudaEvent t start, stop;
float time;

cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord( start, 0 );
kernel<<<grid,threads>>> ( d_odata, d_idata, size_x, size_y,
                           NUM_REPS);
cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop );

cudaEventElapsedTime( &time, start, stop );
cudaEventDestroy( start );
cudaEventDestroy( stop );
```

Figura 12 - Exemplo temporizador da API CUDA (NVIDIA, 2011).

A função *cudaEventRecord()* é usada para iniciar e parar eventos no *stream defaul* 0. O *device* irá gravar um *timestamp* para o evento quando ele atinge esse evento no fluxo.

A função *cudaEventElapsedTime()* retorna o tempo decorrido entre o registro do início e do fim. Este valor é expresso em milissegundos. Note que os tempos são medidos no relógio da GPU, portanto o tempo obtido é independente do sistema operacional, o que é uma grande vantagem.

Neste projeto o tempo é essencial para medir e comparar o desempenho do paralelismo usando CUDA e usando apenas a CPU com um algoritmo sequencial. A partir desses dados, serão concluídas as medidas de desempenho.

3.5. Otimização de Memória

A otimização da memória é uma das áreas mais importantes para o desempenho. O objetivo é maximizar o uso do *hardware*, maximizando a largura de banda. Esta seção discute os vários tipos de memória no *host* e no *device* e a melhor forma de configurar os itens de dados para usar a memória de forma eficaz.

3.5.1. Transferência de dados entre Host e Device

O pico de banda larga entre a memória do *device* e a GPU é muito maior do que o pico entre a memória do *host* e a memória do *device*. A alta prioridade seria minimizar a transferência de dados entre o *host* e o *device* mesmo que signifique a execução de alguns *kernels* no *device* sendo que seriam mais rápidos executados no *host* (CPU) ou vice-versa.

Estruturas intermediárias de dados devem ser criadas, operadas e destruídas na memória do *device*, sem nunca terem sido mapeadas pelo *host* ou copiadas para a memória do *host*. Além disso, devido a sobrecarga da transferência, muitas pequenas transferência de lotes em uma maior transferência tem um desempenho melhor do que fazer cada transferência separadamente.

Mesmo fazendo parte do mesmo barramento, o *host* e o *device* devem evitar transferências de dados excessivas, pois congestionaria o barramento e perderia desempenho. O acesso do *device* em sua própria memória é muito mais rápido que o acesso do *device* na memória do *host*, portanto a quantidade e a velocidade da memória do *device* são importantíssimas para o desempenho do mesmo. É preferível fazer uma única transferência de um pacote enorme de dados, do que fazer várias transferências de pacotes menores de dados.

3.5.2. Cópia Zero

Cópia zero é um recurso que foi adicionado na versão 2.2 do *Toolkit* CUDA (NVIDIA, 2011). Ele permite que as *threads* da GPU acessem diretamente a memória do *host*. Para este fim, requer memória

mapeada (não-paginável). Em GPUs integradas (GPUs para notebook), a memória mapeada é sempre um ganho de desempenho, pois evita a cópias sem sentido.

Em GPUs discretas, a memória mapeada é vantajosa apenas em determinados casos. Porque os dados não são armazenados no *cache* do *device*, memórias mapeadas devem ser lidas ou escritas apenas uma vez. A figura 13 mostra como cópia zero é tipicamente configurada.

```
float *a_h, *a_map;
...
cudaGetDeviceProperties(&prop, 0);
if (!prop.canMapHostMemory)
     exit(0);
cudaSetDeviceFlags(cudaDeviceMapHost);
cudaHostAlloc(&a_h, nBytes, cudaHostAllocMapped);
cudaHostGetDevicePointer(&a_map, a_h, 0);
kernel<<<gridSize, blockSize>>>(a_map);
```

Figura 13 - Configurando a Cópia zero Host (NVIDIA, 2011).

Neste código, o campo *canMapHostMemory()* da estrutura, retornado pela função *cudaGetDeviceProperties()* é usado para verificar se o *device* suporta mapear a memória do *host* para o espaço de endereço do *device*. Mapeamento da memória *Page-locked* é ativada por *cudaSetDeviceFlags()* com *cudaDeviceMapHost.* Nota-se que *cudaSetDeviceFlags()* deve ser chamado antes da criação de um *device* ou fazer uma chamada CUDA que necessite deste estado.

A cópia zero é um recurso muito bem utilizado pelo CUDA, onde não há copia da memora do *host* para a memória do *device* para a execução do processo, o *device* acessa diretamente a posição de memória do *host* que precisa para executar o processo. Um recurso muito mais rápido, econômico e simples de programar.

3.5.3. Espaços de memória do Device

O *device* CUDA usa vários espaços de memória, que tem características diferentes que refletem os diferentes usos em aplicações CUDA. Estes espaços de memória incluem global, local, textual, compartilhada, e registradores, como mostra a Figura 14.

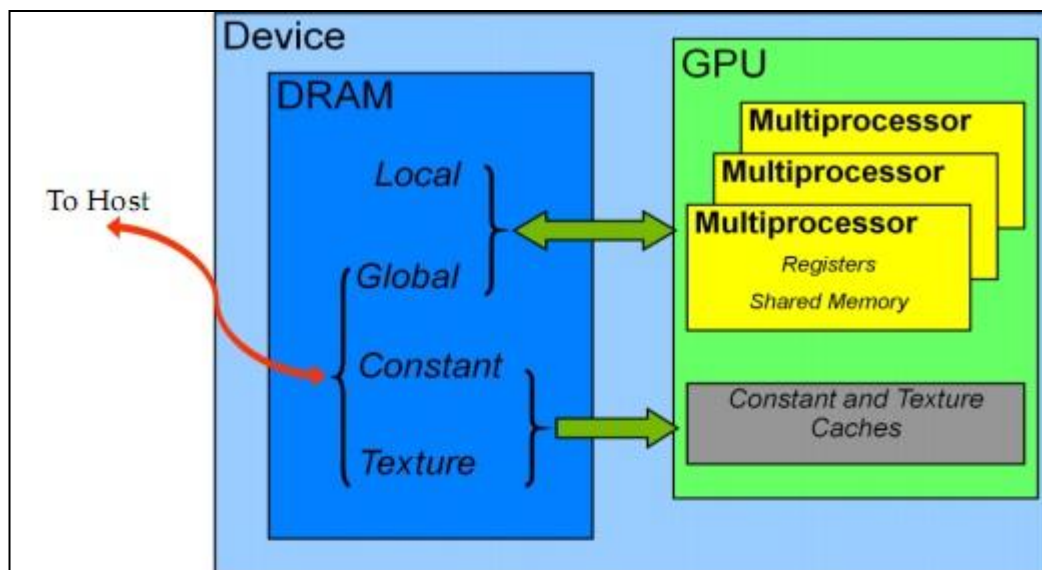

Figura 14 - Espaços de memória do Device (NVIDIA, 2011).

Desses espaços de memória diferentes, memória global e memória de textura são os mais abundantes. Global, local e de textura têm a maior quantidade de acesso, seguida da memória constante, registradores e memória compartilhada.

Alocação e deslocação de memória do *device* são feitas via *cudaMalloc()* e *cudaFree()* e são operações que gastam muito tempo, portanto a memória do *device* deve ser reutilizada e/ou sub-alocados pela aplicação, sempre que possível para minimizar o impacto da atribuição no desempenho geral.

A programação em CUDA é muito similar a programação em linguagem C, percebe-se que a alocação se faz utilizando a função *cudaMalloc()* e em C é utilizada *Malloc()*. CUDA foi baseada em C, portanto quem domina a linguagem C não terá muito problemas em CUDA.

O espaço de memória global, constante e de textura do *device* permitem o *host* acessar, fazendo uma cópia para a memória do *host*. Já os espaços de memória local, apenas o *device* tem acesso e pode usar.

CAPITULO 4 – DESCRIÇÃO DO PROJETO

Testes foram feitos para analisar o desempenho de CUDA através do tempo tirado de cada ordenação de vetores de tamanhos diferentes, utilizando o método de ordenação *Radix Sort*, e comparando com o tempo tirado com os mesmos tamanhos de vetores para o algoritmo em C. Mas para que os testes fossem feitos, foi preciso compila e executar o algoritmo em CUDA e para isso precisou instalar seus drives. As próximas seções explicam como foi feita a instalação.

4.1. Download e Instalação CUDA

Para a utilização de CUDA o programador necessita de um computador com placa de vídeo da marca NVIDIA e precisa instalar seus *drivers*. A seguir será mostrada a instalação dos *drivers* NVIDIA:

Entrou-se no site http://www.Nvidia.com.br, clicou no *link* TECNOLOGIAS e escolheu a opção CUDA. Após a página ter sido carregada clicou-se no *link* ACESSAR CUDA ZONE, outra página foi carregada e clicou-se no *link* CUDA-Downloads. Já na área CUDA-Downloads entrou no *link* escrito "*Get Lastet CUDA Toolkit Production Release*".

Os *drivers* estarão à disposição para *download* na área citada a cima, porém é necessário que o programador saiba em que sistema operacional ele estará programando. Neste projeto foi usado o Mac OS X 32*bits*.

Após a escolha do sistema operacional foi feito o *download* do item *Developer drivers*, CUDA *Toolkit* e GPU *Computing* SDK. O *Developer drivers* tem como função fazer o sistema operacional reconhecer a placa de vídeo como um *hardware* para seu uso. O CUDA *Toolkit* é um pacote que contém compilador C/C++, bibliotecas CUDA e documentação. O compilador C/C++ é necessário para reconhecer e compilar a linguagem C junto ao pacote de bibliotecas CUDA e interpretá-los.

Após a realização dos *downloads* é feita a instalação. O *Developer drivers* foi instalado primeiro e logo após o CUDA *Toolkit* e o GPU *Computing* SDK. Depois da instalação dos *drivers* e da biblioteca do CUDA, definiu-se as variáveis de ambiente às bibliotecas instaladas. Em seguida, foi aberto o terminal e digitou-se dois comandos:

- export PATH =/usr/local/cuda/Bin:$PATH
- export DYLD_LIBRARY_PATH=/usr/local/cuda/lib/:$DYLD_LIBRARY_PATH

As variáveis de ambiente receberam as bibliotecas do CUDA, portanto foram reconhecidas automaticamente. Após o termino da instalação foi verificado, através do comando "kextstat | grep –i cuda" digitado no terminal se foi concluída com sucesso e foi mostrada uma mensagem como mostra a Figura 15.

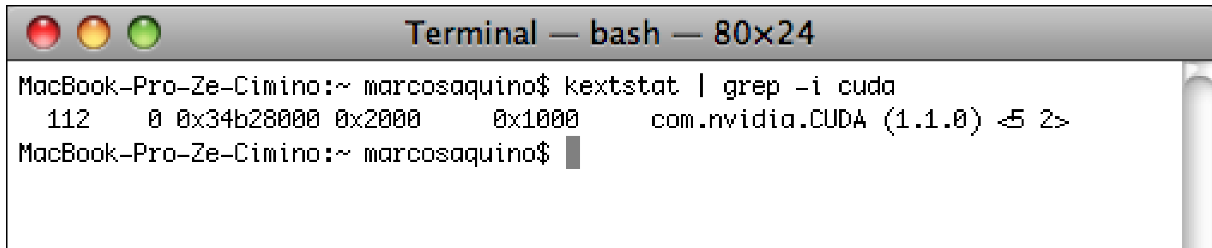

Figura 15 - Versão do driver CUDA instalado (Própria).

4.2. Compilando CUDA

Para testar alguns exemplos de paralelismo usando CUDA, foram usados os vários exemplos que vieram juntos com o *GPU Computing SDK.* Esses exemplos podem ser encontrados no diretório "*/Developer/GPU Computing/C/src* " , são códigos abertos e podem ser modificados.

A verificação do paralelismo usando CUDA foi feita compilando os exemplos do pacote *GPU Computing SDk,* usando o comando "*make*" no terminal. Mas para que isso fosse possível o terminal deve estar apontando para o diretório "*/Developer/GPU Computing/C"*. Após usar o comando "*make*", todos os códigos que haviam no diretório "*/Developer/GPU Computing/C/src* " foram compilados e arquivos executáveis foram gerados. Esses arquivos executáveis podem ser encontrados no diretório "*/Developer/GPU Computing/C/bin/darwin/release* ". A figura 16 mostra um exemplo de paralelismo entre a CPU e a GPU, onde demonstra o sincronismo entre elas.

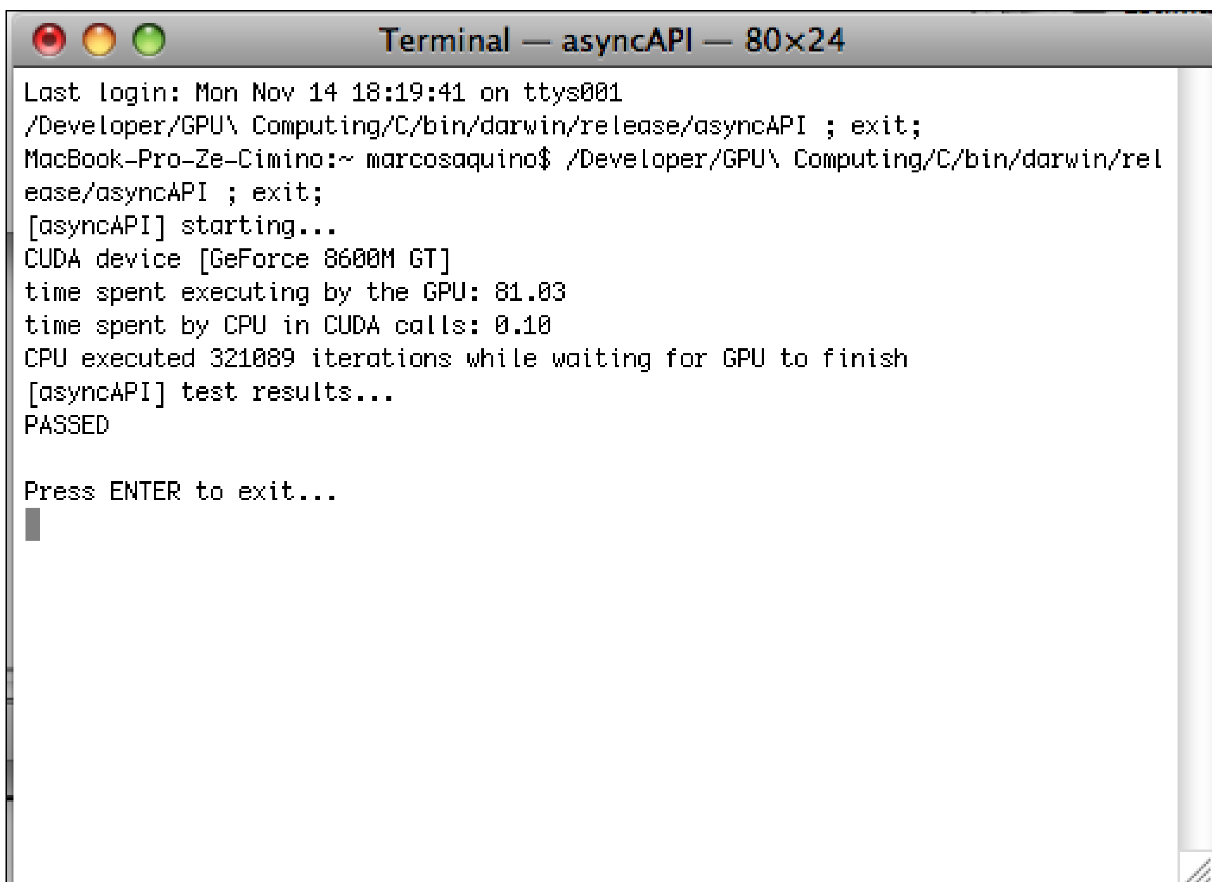

Figura 16 - CPU esperando resultados da GPU (Própria).

4.3. Algoritmos e Métodos

Para a realização de testes de desempenho, neste projeto foi usado um algoritmos de ordenação de vetores chamado *Radix Sort.* O *Radix Sort* é um algoritmo de ordenação por distribuição que ordena com base nos dígitos de um número, prioriza (inicia por) dígitos menos significativos (Mello, 2002). O algoritmo usado é uma implementação do *Radix Sort* com a utilização de CUDA para haver o paralelismo dos dados, ele foi desenvolvido pela empresa NVIDIA e possui a extensão *.cu*. Foi usado também para testes e análises de resultados o algoritmo *Radix Sort* implementado na linguagem C, de forma sequencial para que seja executado na CPU. A análise do desempenho foi feita através do tempo de execução, tirada em segundos, de cada algoritmo, comparando-os.

CAPITULO 5 – ANÁLISES, TESTES E RESULTADOS

Para analisar o desempenho do paralelismo entre a CPU e a GPU, foram feitos testes no algoritmo *Radix Sort* com CUDA. Os testes foram feitos mudando o tamanho do vetor, compilando o algoritmo e tirando seu tempo de execução. Para cada tamanho de vetor o programa foi executado quatro vezes e cada execução foi obtido tempos diferentes e registrados para fins estatísticos.

Os testes foram feitos com vetores de 5, 10, 100, 1.000, 10.000, 100.000, 1.000.000, 10.000.000 e 15.000.000 de posições, e como mostra a tabela 1, seus resultados foram:

Quantidade de Elementos do vetor	Tempo em segundos				Média
	Teste 1	Teste 2	Teste 3	Teste 4	
15.000.000	0.59044	0.59018	0.59086	0.59043	0.5904775
10.000.000	0.39336	0.39410	0.39230	0.39304	0.3932000
1.000.000	0.04340	0.04380	0.04392	0.04364	0.0436900
100.000	0.00550	0.00567	0.00554	0.00563	0.0055850
10.000	0.00157	0.00156	0.00157	0.00159	0.0015725
1.000	0.00086	0.00101	0.00087	0.00091	0.0009125
100	0.00075	0.00071	0.00073	0.00073	0.0007300
10	0.00070	0.00070	0.00078	0.00083	0.0007525
5	0.00073	0.00070	0.00072	0.00072	0.0007175

Tabela 1- Tempo de execução da ordenação de vetores com tamanhos diferentes. Linguagem C + CUDA (Própria).

Pode-se perceber que os quatro testes feitos com cada vetor teve uma média constante, como por exemplo, o de 5 posições com os tempos 0.00073s, 0.00070s, 0.00072s e 0.00072s e com uma média de 0.0007175s e o vetor com 10 posições obteve o tempo de 0.00070s, 0.00070s, 0.00078s e 0.00083s, com uma média de 0.0007525s. A média do vetor com 10 posições foi maior devido a maior quantidade de números no vetor para serem ordenados.

O tempo de transferência de dados entre a CPU e a GPU, feita através de barramentos, não foi considerado para estes testes, apenas o tempo de execução para a ordenação dos vetores.

Para comparação de desempenho da tecnologia CUDA, foi usado neste projeto um algoritmo *Radix Sort* implementado na linguagem C (Silva, 2011), e foi modificado para medir o tempo de execução. A tabela 2 mostra seus resultados.

Quantidade de Elementos do vetor	Tempo em segundos				Média
	Teste 1	Teste 2	Teste 3	Teste 4	
15.000.000	x	x	x	x	x

10.000.000	x	x	x	x	x
1.000.000	1.725130	1.729930	1.724790	1.727920	1.7269425
100.000	0.176377	0.175683	0.176774	0.175771	0.1761512
10.000	0.017701	0.017718	0.017973	0.017684	0.0177690
1.000	0.001846	0.001877	0.001902	0.001854	0.0018697
100	0.000324	0.000322	0.000345	0.000325	0.0003290
10	0.000129	0.000125	0.000124	0.000125	0.0001257
5	0.000115	0.000112	0.000112	0.000122	0.0001152

Tabela 2 - Tempo de execução da ordenação de vetores com tamanhos diferentes. Linguagem C (Própria).

O teste feito com o algoritmo implementado na linguagem C, sequencial, não suportou mais de 1.000.000 elementos no vetor. As marcas representadas com *x* na tabela 2 representam falta de memória para a execução do programa.

Os algoritmos usados para os teste são implementações da técnica de ordenação de vetores *Radix Sort*, mesmo eles contendo o mesmo método de ordenação, algumas diferenças foram levadas em consideração, como quantidade de funções e transferência de dados entre o *host* e o *device* .

Os resultados foram comparados entre as médias, como mostram os gráficos 1, 2, 3, 4, 5, 6 e 7.

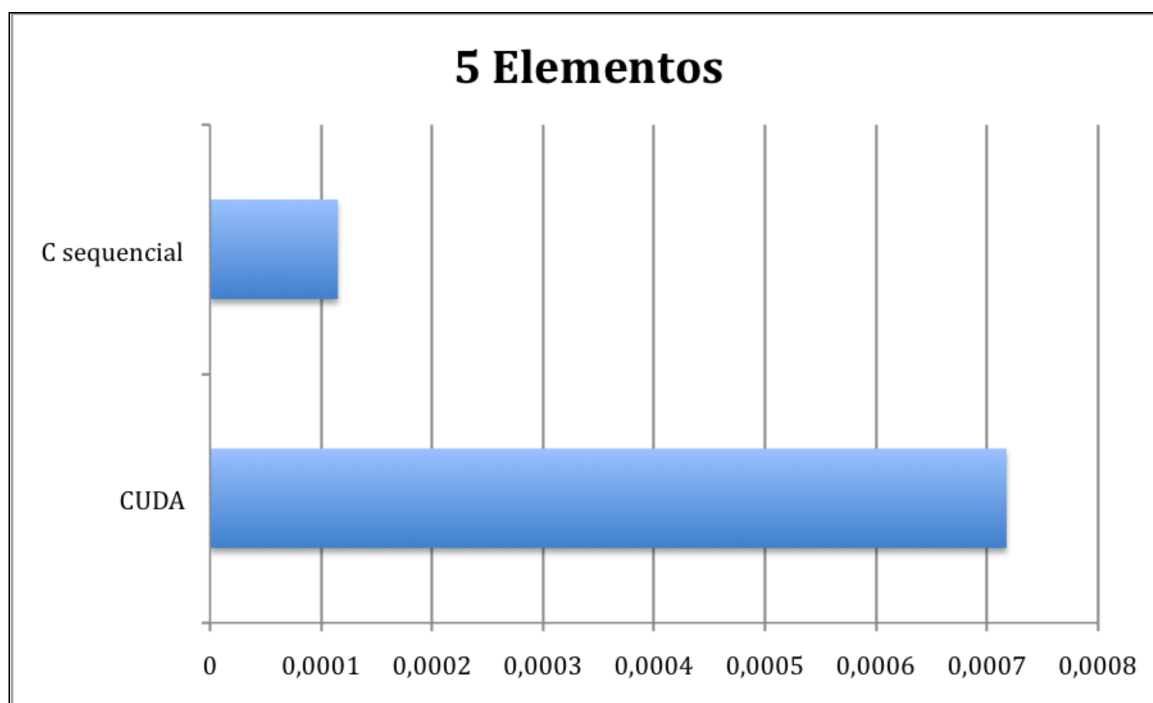

Gráfico 1 - 5 elementos (Própria).

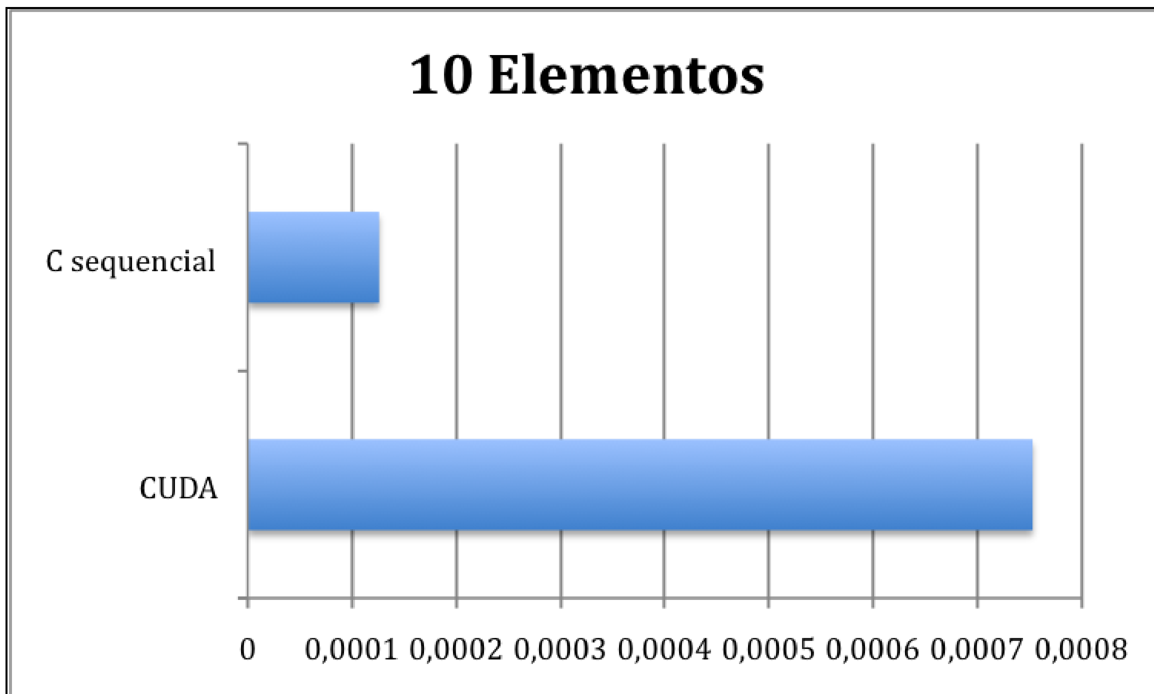

Gráfico 2 - 10 Elementos (Própria).

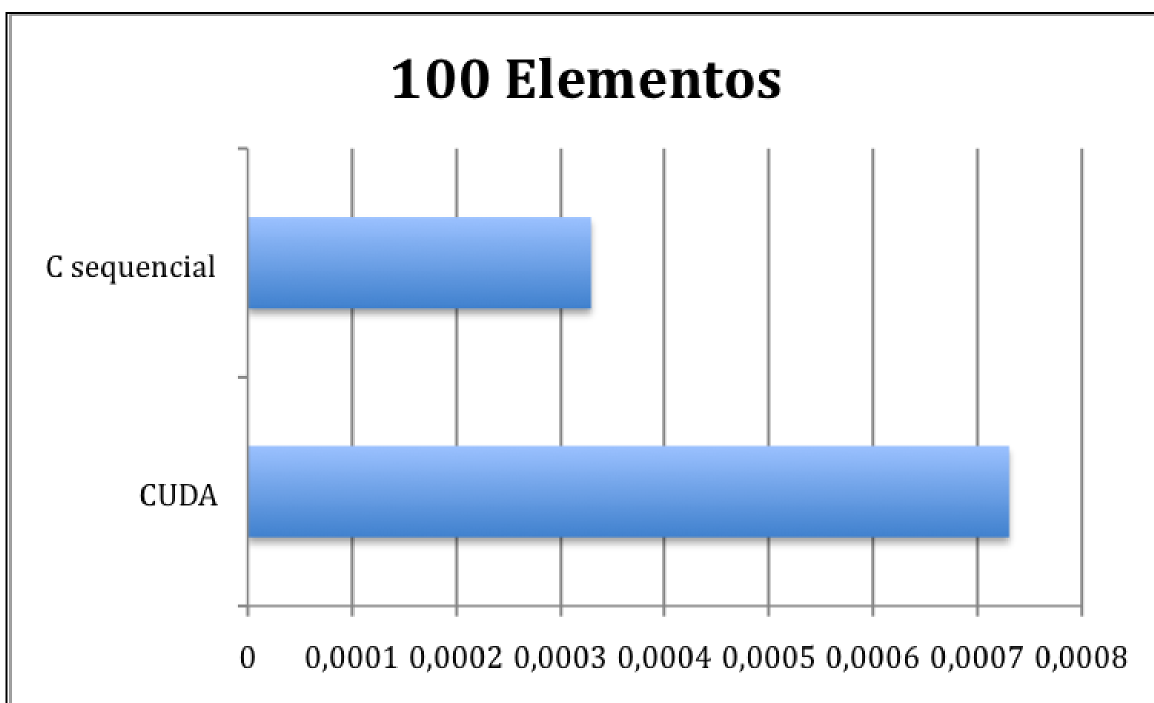

Gráfico 3 - 100 Elementos (Própria).

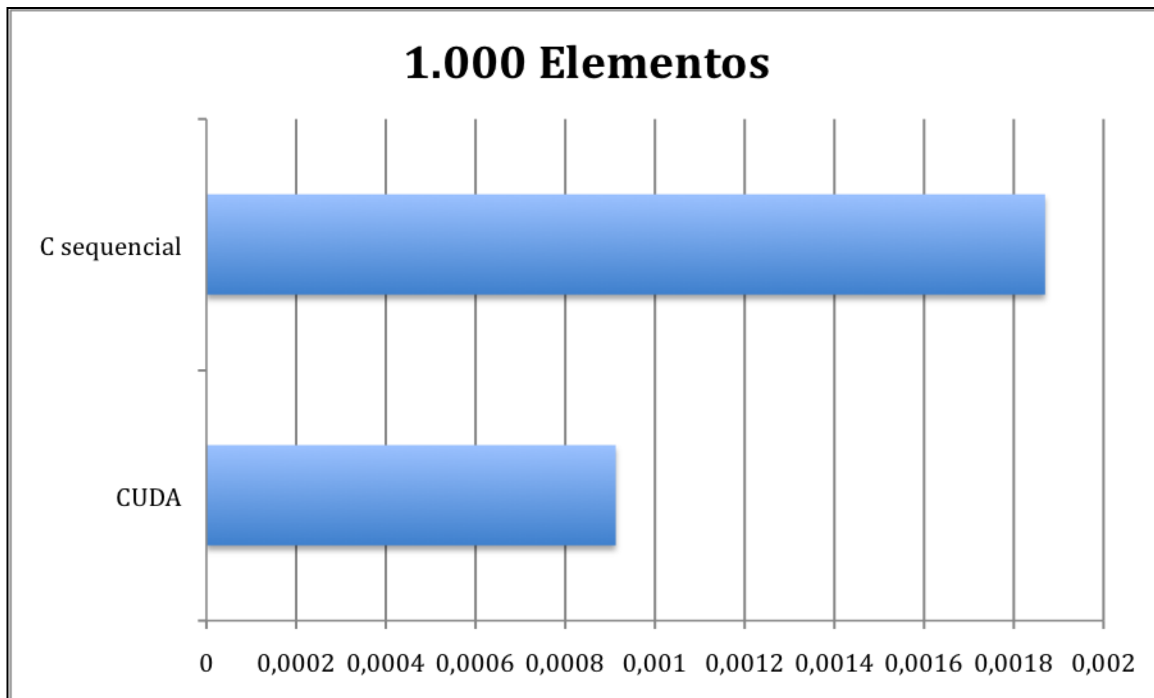

Gráfico 4 - 1.000 Elementos (Própria).

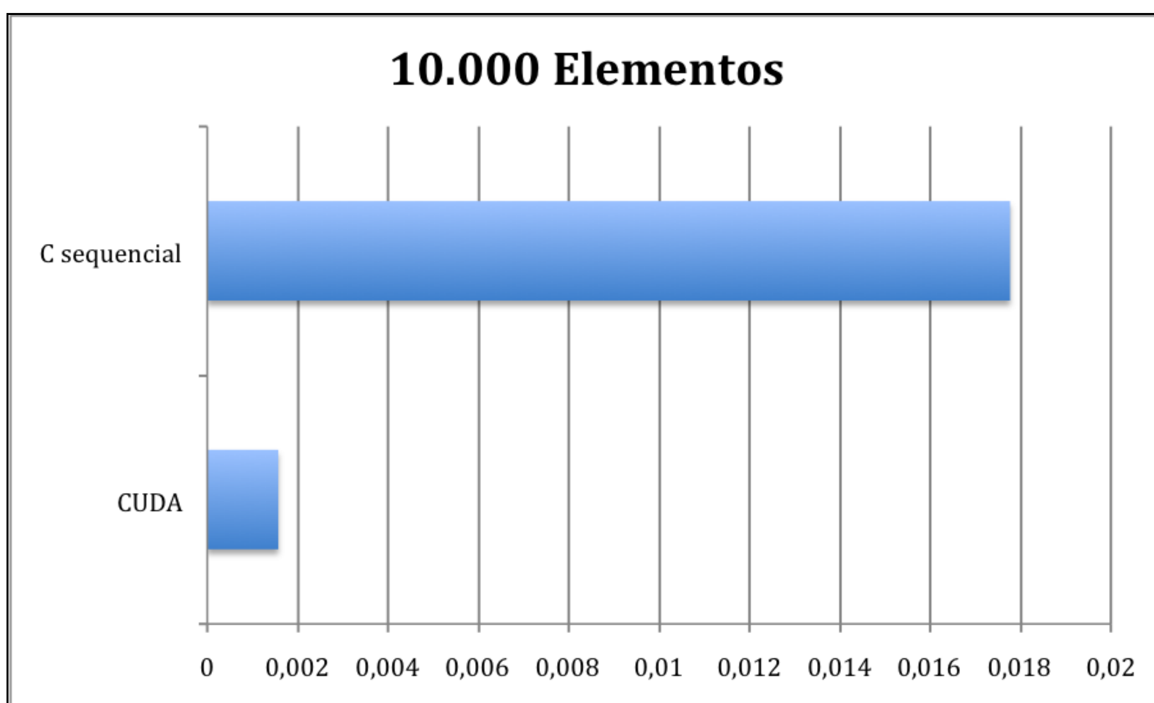

Gráfico 5 - 10.000 Elementos (Própria).

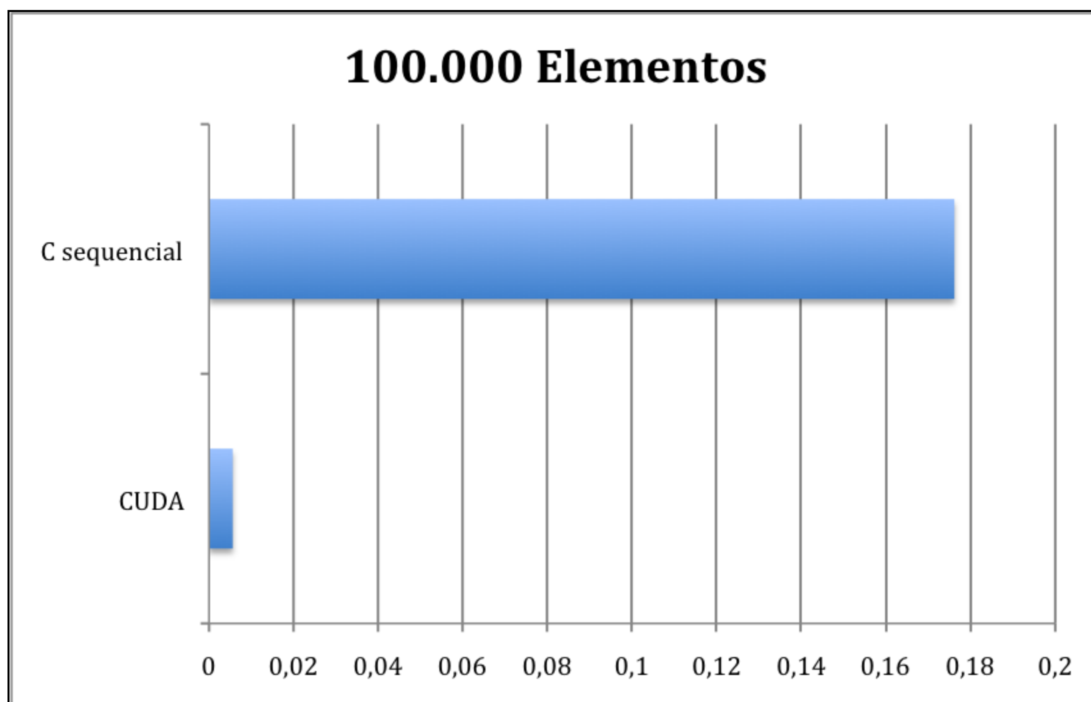

Gráfico 6 - 100.000 Elementos (Própria).

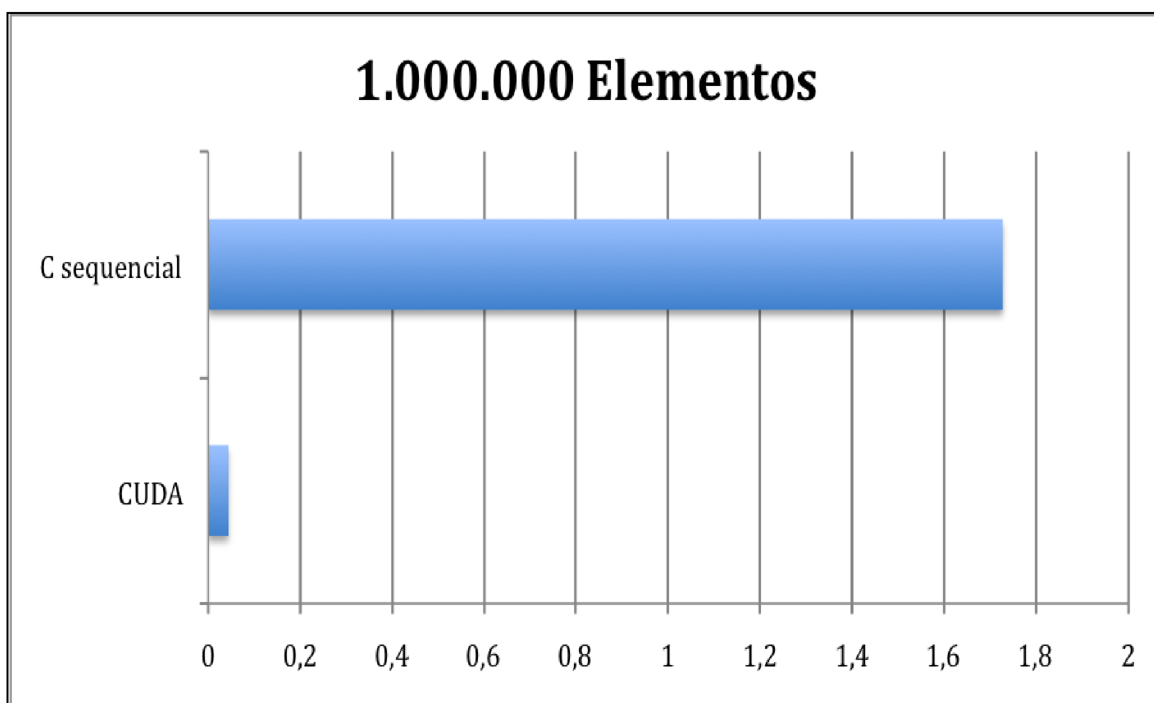

Gráfico 7 - 1.000.000 Elementos (Própria).

O algoritmo C sequencial obteve um melhor desempenho com vetores menores, de 5 elementos, 10 elementos e 100 elementos. A partir de 1.000 elementos seu desempenho caiu e seu tempo aumentou, e o paralelismo de CUDA se manteve equilibrado, e obteve um bom desempenho. Outra vantagem do CUDA nestes testes foi a quantidade de elementos do vetor que chegou até aproximadamente 15.000.000 de elementos, e o C sequencial chegou apenas até 1.000.000 de elementos.

O desempenho CUDA foi inferior com vetores menores ao fato de que houve a troca de

informações entre o *host* e o *device* fazendo com que seu tempo de execução aumentasse mesmo tendo poucos elementos para ordenar. Já o desempenho do algoritmo C sequencial foi superior com menos elementos no vetor por não necessitar das trocas de informações, ele apenas faz a comunicação entre a memória principal do computador e executa sequencialmente seu código. Porém o desempenho de CUDA foi superior com uma grande quantidade de elementos, maior que 1.000, devido a sua grande capacidade de executar várias *threads* ao mesmo tempo e seu grande poder de processamento, mesmo havendo a troca de dados entre o *host* e o *device*, seu desempenho foi melhor.

CAPITULO 6 – CONCLUSÃO

De acordo com os testes realizados, pode-se concluir que quando se trata de uma aplicação de uso geral rápida, como a ordenação dos vetores com 5, 10 e 100 elementos, o uso do paralelismo perdeu desempenho comparado com o seqüencial. A CPU executou essas ordenações em menos tempo, porém uso do paralelismo executou a ordenação de grandes vetores em pouco tempo, devido o uso excessivo de várias *thread*, assim mostrando seu grande poder computacional. Conclui-se também que o uso do paralelismo entre CPU e GPU foi mais eficaz devido a GPU possuir memória própria, não precisando ficar acessando a memória principal do computador, assim aumentando seu desempenho.

A GPU em auxílio para aceleração de aplicações de propósito geral pode ser uma grande solução para quem quer maior desempenho, mas cabe ao programador decidir se há necessidade do paralelismo, como por exemplo, o *Radix Sort* para poucos elementos, ou multiplicações de matrizes de grandes tamanhos.

REFERÊNCIAS

AUGENSTEIN, M. J.; LANGSAM, Y.; TENENBAUM, A. M.. **Estruturas de dados usando C**. São Paulo, Makron Books, 884p. 2004.

CALISKAN, K. **Gpu Opencl vs Cuda vs Arbb.** Disponível em: <http://www.keremcaliskan.com/gpgpu-opencl-vs-cuda-vs-arbb>. Acesso em 13 abril 2011.

CARDOSO, B., ROSA, S. R. A. S., FERNANDES, T. M.. **Multicore.** Disponível em: <http://www.ic.unicamp.br/~rodolfo/Cursos/mc722/2s2005/Trabalho/g07-multicore.pdf>. Acesso em 26 julho 2011.

IC UFF. **Uma Pequena História da Computação.** Disponível em: <http://www.ic.uff.br/~otton/graduacao/informaticaI/computadores_por_imagens.html>. Acesso em: 28 abril 2011.

INTEL. 4004 Single Chip 4-Bit. P-Channel Microprocessor. Disponível em: <http://www.intel.com/Assets/PDF/DataSheet/4004_datasheet.pdf>. Acesso em: 17 março 2011.

INTEL. Intel High-Performance Consumer Desktop Microprocessor Timeline. Disponível em: <http://www.intel.com/pressroom/kits/core2duo/pdf/microprocessor_timeline.pdf>. Acesso em 17 março 2011.

INTEL. **A história dos processadores, desde ENIAC até Nehalem.** 2011. Disponível em: <http://cache-www.intel.com/cd/00/00/40/17/401775_401775.pdf>. Acesso em: 28 março 2011.

KIRK, D. B. ; HWU, M. W.. **Programando para Processadores Paralelos.** 1ª ed. Campus, 2010.

MELLO, R. S. **Estrutura de Dados.** 2002. Disponível em: < http://www.inf.ufsc.br/~ronaldo/ine5384/15-OrdenacaoDados.pdf>. Acesso em: 7 outubro 2011.

MUSEU DO COMPUTADOR. **Primeiro Processador do Mundo.** Disponível: <http://www.museudocomputador.com.br/encipro.php>. Acesso em: 28 março 2011.

MUSEU DO COMPUTADOR. **Segunda Geração de Processadores.** Disponível: <http://www.museudocomputador.com.br/encipro.php>. Acesso em: 28 março 2011.

NVIDIA. **CUDA Toolkit 4.0.** Disponível em: <http://developer.Nvidia.com/cuda-toolkit-40>. Acesso em: 15 agosto 2011.

PITANGA, M. **Computação em Cluster**. Brasport. 344p. 2004.

ROCHA, R. Programação Paralela e Distribuída. 2008. Disponível em: < http://www.dcc.fc.up.pt/~ricroc/aulas/0708/ppd/apontamentos/fundamentos.pdf>. Acesso em 28 agosto 2011.

SILVA, J. C. M.. Algoritmo de Ordenação Radix Sort [C/C++ Script]. Disponível em: <http://www.vivaolinux.com.br/script/Algoritmo-de-Ordenacao-Radix>. Acesso em: 01 novembro 2011.

STALLINGS, W.. **Arquitetura e Organização de Computadores**.5ª Edição. Prentice Hall. 2003.

TANENBAUM, A. S.; VAN STEEN, M.. **Sistemas distribuídos**. 2ª ed. São Paulo, Pearson, 402p. 2008.

TEC MUNDO. **A história dos processadores.** Disponível em: <http://www.tecmundo.com.br/2157-a-historia-dos-processadores.htm>. Acesso em: 20 agosto 2011.

TOM'S HARDWARE. **The King Of Efficiency: The Intel Pentium M 780.** Disponível em: <http://www.tomshardware.com/reviews/dual-core-intel-processors-low,1247-9.html>. Acesso em: 28 abril 2011.

ANEXO

ANEXO A -

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <thrust/sequence.h>
```

```
#include <thrust/random.h>
#include <thrust/generate.h>
#include <thrust/detail/type_traits.h>

#include <cutil_inline.h>

#include <shrUtils.h>
#include <shrQATest.h>
#include <algorithm>
#include <time.h>
#include <limits.h>

template <typename T, bool floatKeys>
bool testSort(int argc, char **argv)
{
    int cmdVal;
    int keybits = 32;

    unsigned int numElements = 104857;
    bool keysOnly = (cutCheckCmdLineFlag(argc, (const char**)argv, "keysonly") == CUTTrue);
    bool quiet = (cutCheckCmdLineFlag(argc, (const char**)argv, "quiet") == CUTTrue);

    if( cutGetCmdLineArgumenti( argc, (const char**)argv, "n", &cmdVal) )
    {
        numElements = cmdVal;
    }

    if( cutGetCmdLineArgumenti( argc, (const char**)argv, "keybits", &cmdVal) )
    {
        keybits = cmdVal;

        if (keybits <= 0) {
                        printf("Error: keybits must be > 0, keybits=%d is invalid\n", keybits);
                        exit(0);
        }
    }
    unsigned int numIterations = (numElements >= 16777216) ? 10 : 100;
    if ( cutGetCmdLineArgumenti(argc, (const char**) argv, "iterations", &cmdVal) )
    {
        numIterations = cmdVal;
    }

    if( cutCheckCmdLineFlag(argc, (const char**)argv, "help") )
    {
        shrLog("Command line:\nradixsort_block [-option]\n");
        shrLog("Valid options:\n");
        shrLog("-n=<N>        : number of elements to sort\n");
        shrLog("-keybits=bits : keybits must be > 0\n");
        shrLog("-keysonly     : only sort an array of keys (default sorts key-value pairs)\n");
        shrLog("-float        : use 32-bit float keys (default is 32-bit unsigned int)\n");
        shrLog("-quiet        : Output only the number of elements and the time to sort\n");
        shrLog("-help         : Output a help message\n");
```

```
        exit(0);
    }

    if (!quiet)
        shrLog("\nSorting %d %d-bit %s keys %s\n\n", numElements, keybits, floatKeys ? "float" : "unsigned
int", keysOnly ? "(only)" : "and values");

    int deviceID = -1;
    if (cudaSuccess == cudaGetDevice(&deviceID))
    {
        cudaDeviceProp devprop;
        cudaGetDeviceProperties(&devprop, deviceID);
        unsigned int totalMem = (keysOnly ? 2 : 4) * numElements * sizeof(T);
        if (devprop.totalGlobalMem < totalMem)
        {
            shrLog("Error: not enough memory to sort %d elements.\n", numElements);
            shrLog("%d bytes needed, %d bytes available\n", (int) totalMem, (int) devprop.totalGlobalMem);
            exit(0);
        }
    }

    thrust::host_vector<T> h_keys(numElements);
    thrust::host_vector<T> h_keysSorted(numElements);
    thrust::host_vector<unsigned int> h_values;
    if (!keysOnly)
        h_values = thrust::host_vector<unsigned int>(numElements);

    // Fill up with some random data
    thrust::default_random_engine rng(clock());
    if (floatKeys)
    {
        thrust::uniform_real_distribution<float> u01(0, 1);
        for(int i = 0; i < (int)numElements; i++)
            h_keys[i] = u01(rng);
    }
    else
    {
        thrust::uniform_int_distribution<unsigned int> u(0, UINT_MAX);
        for(int i = 0; i < (int)numElements; i++)
            h_keys[i] = u(rng);
    }

    if (!keysOnly)
        thrust::sequence(h_values.begin(), h_values.end());

    // Copy data onto the GPU
    thrust::device_vector<T> d_keys;
    thrust::device_vector<unsigned int> d_values;

    // run multiple iterations to compute an average sort time
    cudaEvent_t start_event, stop_event;
    cutilSafeCall( cudaEventCreate(&start_event) );
```

```
    cutilSafeCall( cudaEventCreate(&stop_event) );

    float totalTime = 0;
    for(unsigned int i = 0; i < numIterations; i++)
    {
        // reset data before sort
        d_keys= h_keys;
                if (!keysOnly)
                    d_values = h_values;

                cutilSafeCall( cudaEventRecord(start_event, 0) );

        if(keysOnly)
            thrust::sort(d_keys.begin(), d_keys.end());
        else
            thrust::sort_by_key(d_keys.begin(), d_keys.end(), d_values.begin());

                cutilSafeCall( cudaEventRecord(stop_event, 0) );
                cutilSafeCall( cudaEventSynchronize(stop_event) );

                float time = 0;
                cutilSafeCall( cudaEventElapsedTime(&time, start_event, stop_event));
                totalTime += time;
    }
    totalTime /= (1.0e3f * numIterations);
    shrLogEx(LOGBOTH | MASTER, 0, "radixSort, Throughput = %.4f MElements/s, Time = %.5f s, Size =
%u elements\n",
                1.0e-6f * numElements / totalTime, totalTime, numElements);

    CUT_CHECK_ERROR("after radixsort");

    // Get results back to host for correctness checking
    thrust::copy(d_keys.begin(), d_keys.end(), h_keysSorted.begin());
    if (!keysOnly)
        thrust::copy(d_values.begin(), d_values.end(), h_values.begin());

    CUT_CHECK_ERROR("copying results to host memory");

    // Check results
    bool bTestResult = thrust::is_sorted(h_keysSorted.begin(), h_keysSorted.end());

    cutilSafeCall( cudaEventDestroy(start_event) );
    cutilSafeCall( cudaEventDestroy(stop_event) );

    if ( !bTestResult  && !quiet) {
        return false;
    }
    return bTestResult;
}

int main(int argc, char **argv)
{
```

```
    shrQAStart(argc, argv);

    // Start logs
    shrSetLogFileName ("radixSort.txt");
    shrLog("%s Starting...\n\n", argv[0]);

    cutilDeviceInit(argc, argv);

    bool bTestResult = false;

    if( cutCheckCmdLineFlag( argc, (const char**)argv, "float") )
        bTestResult = testSort<float, true>(argc, argv);
    else
        bTestResult = testSort<unsigned int, false>(argc, argv);

    shrQAFinishExit(argc, (const char **)argv, bTestResult ? QA_PASSED : QA_FAILED);
}
```