Curso de Ciência da Computação
Universidade Estadual de Mato Grosso do Sul

ALGORITMOS EM GPU

ENNERY SIMILIEN

MSc. André Chastel Lima (Orientador)

Dourados - MS
2021

ALGORITMOS EM GPU

Ennery Similien

Este exemplar corresponde à redação final da monografia da disciplina Projeto Final de Curso devidamente corrigida e defendida por Ennery Similien e aprovada pela Banca Examinadora, como parte dos requisitos para a obtenção do título de Bacharel em Ciência da Computação.

Dourados, 22 de novembro de 2021.

Prof. MSc. André Chastel Lima (Orientador)

S61a Similien, Ennery
Algoritmos em GPU / Ennery Similien. – Dourados, MS: UEMS, 2021.
102p.

Monografia (Graduação) – Ciência da Computação – Universidade Estadual de Mato Grosso do Sul, 2021.
Orientador: Prof. MSc. André Chastel Lima

1. Algoritmos em GPU; 2. Programação de GPU; 3. CUDA I. Lima, André Chastel II. Título

CDD 23.ed. – 005.1

Curso de Ciência da Computação
Universidade Estadual de Mato Grosso do Sul

ALGORITMOS EM GPU

Ennery Similien

Novembro de 2021

Banca Examinadora:

Prof. MSc. André Chastel Lima (Orientador)
Área de Computação – UEMS

Prof. Dra. Raquel Marcia Müller
Área de Computação – UEMS

Prof. Dr. Rubens Barbosa Filho
Área de Computação – UEMS

A minha mãe Ymeriane Viljean, ao meu pai Emmanuel Similien, a minha filha Nailah Adassa N. Similien, aos meus irmãos e a minha namorada, amores eternos.

AGRADECIMENTOS

A todos que, diretamente ou indiretamente, contribuíram para o sucesso desse estudo. Agradecimento especial a meu prezado orientador e professor, Sr. Prof. André Chastel Lima, pela paciência e pela oportunidade; pelo seu constante acompanhamento durante toda a trajetória; na verdade, foi mais do que um orientador, foi um companheiro de estrada. Aos meus colegas que, desde o primeiro dia na faculdade, me apoiaram, me direcionaram, me ensinaram cada vez mais, produzindo enfim, o que eu sou hoje. Aos meus professores, não tenho palavras para descrever o que foram para mim nesses quatro (4) anos de estudo; as suas atuações durante esse período me fizeram concluir que: “Ser professor é, antes de ser um trabalho bem remunerável, ser o guardião do futuro do seu país e da humanidade”.

A minha família, não há como agradecer infinitamente, mas quero mesmo assim dizer que, vocês são a indústria por trás desse produto. Um agradecimento especial a minha mãe Dona Ymeriane Viljean, ao meu pai Emmanuel Similien, aos meus irmãos, à minha namorada, e à minha filha Nailah A. N. Similien por terem acreditados em mim e me apoiados cegamente em tudo. A todos os outros atores dos bastidores, eu agradeço e serei sempre muito grato.

RESUMO

Conforme o uso da GPUs for crescendo na indústria, universidades e empresas que desejam ter mais poder de processamento, tornam-se incrementos valiosos os conhecimentos em programação desses dispositivos. Tendo em vista a necessidade de conteúdos extras aos cursos de Ciência e Engenharia de Computação, e Sistema de Informação; este estudo consiste em um material (tópico) complementar a formação de cientistas e engenheiros da computação, assim com profissionais em sistema de informação. Tendo como abrangência a comunidade dos estudantes da cidade universitária, e focado nos algoritmos de ordenação, o mesmo proporcionar uma base firme em programação CUDA. Este conteúdo proporciona uma base teórica a respeito das GPUs e conceitos importantes a abordagem da programação das mesmas, assim como uma visão interessante sobre a implementação de algoritmos de ordenação com CUDA.

Palavras-chave: Algoritmos em GPU; Programação de GPU; CUDA

SUMÀRIO

LISTA DE FIGURAS

1. INTRODUÇÃO

As bibliografias usadas na maior parte deste capítulo são os livros ***Organização e Projeto de Computadores, e Arquitetura de Computadores*** *de PATTERSON e HENNESSY, apenas parágrafos retirados em outras fontes serão referenciados.*

Há muito tempo, para atender à exigência de sistemas que necessitam muito processamento, os projetistas têm procurado meios de satisfazer o mercado. A ideia base era criar computadores poderosos agrupando computadores menores; isto é, colocar em um único chip vários processadores em paralelo, que hoje chamamos de **multiprocessadores.** Essa abordagem traz ganho de desempenho se o software que os utiliza puder aproveitá-los com eficiência. O problema dessa tentativa é que, pelo tamanho das CPUs, o número de processadores que podem ser alocados em um multiprocessador é limitado, além de que, a maioria dos programas de aplicação não aproveita o paralelismo com eficiência, pois são escritos sequencialmente.

Em contrapartida, uma arquitetura massivamente paralela compensando custo e desempenho mostra mais eficiência em resolver problemas complexos de otimização. Esta é equipada de muitos núcleos e oferece possibilidade de programação de aplicações de propósito geral, são as GPGPUs (*General Purpose Graphic Processing Unit*), isto é, GPUs (*Graphic Processing Unit*) que podem ser usadas e programadas de forma a processar programas que se diferem da sua essência. Desde a sua aparição no mercado, há alguns anos, os pesquisadores têm se interessado em usá-las para cálculos científicos e, entre elas, computação evolutiva que é uma parcela emergente de pesquisa da IA (Inteligência Artificial) que sugere um novo paradigma para solução de problemas inspirado na Seleção Natural (BAUMES; KRUGER; JIMENEZ; COLLET; CORMA, 2011).

Hoje muitos programadores de aplicações científicas e de multimídia estão se ponderando, aproveitando as GPUs e CPUs, isto é, utilizar o extenso paralelismo disponível nas GPUs para acelerar a execução de programas de aplicação que requerem muito processamento.ss

Neste estudo, são abordadas a programação e execução de algoritmos em Unidades de Processamento Gráficas (GPU) justamente porque, as mesmas contam com vários processadores paralelos e muitas threads concorrentes; a sua memória principal é orientada a largura de banda em vez de latência; e em uma GPU, podem ser acomodados muito mais processadores paralelos e threads em comparação a um multiprocessador, garantindo, portanto, alto desempenho.

Contudo, vale ressaltar que, neste estudo, o foco é a programação de GPU para aplicação de uso geral, ou seja, não é observada a programação para o processamento gráfico. Mas então, porque executar algoritmos em GPUs? Qual é o ganho, ou qual é a relação do ganho em speedup na execução de um algoritmo em GPU em comparação a abordagem de processamento de algoritmo no multiprocessador?

1.1 Objetivos e Justificativa

Sendo as GPUs, inicialmente, projetadas para a renderização gráfica em monitores de computadores (e ainda são usadas assim); recentemente, as mesmas são usadas com frequência no processamento de aplicações científicas, engenharia, finanças, entre outras (SANDERS; KANDROT, 2010). Logo, saber programar essas últimas constituem um complemento importante aos profissionais da área. Como objetivo, este estudo visa a pesquisar a respeito das GPUs e produzir um documento que possa auxiliar a comunidade de estudantes em computação (Engenharia ou Ciência) no estudo da programação de Unidades de Processamento Gráfica para processar aplicações de propósito geral. Partindo dos objetivos específicos a seguir:

- Estudar a teoria por trás das GPUs em geral e arquitetura de sistema GPU;
- Aprender a programar as GPUs usando a plataforma CUDA; e
- Comparar a execução de alguns algoritmos de ordenação em GPU com a sua execução em CPU.

Hoje em dia, as empresas estão substituindo grandes processadores ineficazes por vários processadores menores que podem proporcionar um desempenho melhor

por watt ou joule, é a era dos multiprocessadores. Essa abordagem permite que, a cada dia, tenhamos arquiteturas massivamente paralelas. As GPUs fazem parte desta família de multiprocessadores, tornam-se multiprocessadores de uso geral graças a sua programabilidade.

As arquiteturas paralelas proporcionam grandes oportunidades de desempenho a programas de aplicação que requerem um poder de computação imenso. Para aproveitar-se com eficiência o poder dessas máquinas, precisa-se saber como elas trabalham e, também, saber programá-las (MATLOFF, 2011).

A área da computação torna-se mais ampla, novas tecnologias surgem, novas subáreas, entre outros. Com o uso atual da GPUs na aceleração de programas de aplicação, e sabendo que, nas universidades, o tempo de estudo em Ciência ou Engenharia da computação é limitado; então torna-se importante a produção de conteúdos complementares a esses cursos. Logo, estudos como este, servem como complementos na formação de Cientistas e Engenheiros da computação; permitindo-os adquirir conhecimentos importantes fora das disciplinas dos cursos.

1.2 Metodologia

Este estudo faz uso, como método de abordagem, a pesquisa de laboratório, que tem como particularidade a manipulação e o controle do alvo (ou do agente) que está sendo estudado, com perspectiva de relacionar motivos e consequências, através da manipulação de variáveis empregadas pelo investigador (ou pesquisador) na experimentação. Fundamentar-se, geralmente, em uma hipótese, esse tipo de pesquisa busca atender as causas de efeitos produzidos. São feitos, esses experimentos, em laboratório ou ao ar livre, em meios sintéticos ou concretos limitados a tais manipulações.

Por adequação, é empregado neste estudo, o método hipotético dedutivo, isto é, procedimento científico segmento de um dilema, pelo qual se proporciona uma solução indefinida (transitória); esta mesma, faz objeto de crítica com intuito de

melhorá-la, eliminando possíveis inconsistências. Logo, o mesmo, ou seja, o ato de criticar e eliminar erros, se repete; pois dá origem a novos problemas.

O uso do método tipológico é feito para caracterizar o problema e gerar (ou definir) um modelo sobre qual são baseados os trabalhos desenvolvidos, durante este estudo. No processo, por observação direta, são examinados os fatos, individualmente, de maneira intensiva, participante, em laboratório adequado ao escopo do estudo.

Para entender a programação das GPUs, primeiramente, é apresentado um estudo teórico sobre as Unidades de Processamento Gráficas e o paralelismo em nível de hardware utilizado nas mesmas; a CUDA (*Compute Unified Device Architecture*) que é uma plataforma de computação paralela de propósito geral desenvolvida pela NVIDIA para a programação de GPUs; assim como, o pacote de ferramentas CUDA Toolkit, que contém tudo que é necessário para implementar aplicações aceleradas por GPUs, como: bibliotecas, um compilador, ferramentas de desenvolvimento e o ambiente de execução de CUDA (NVIDIA DEVELOPER, 2021). Em seguida, é apresentado um primeiro passo com CUDA. Também é visto como implementar algoritmos de processamento paralelo utilizando a plataforma CUDA. E por fim, são apresentados, implementados e comparados, alguns algoritmos de ordenação em um multiprocessador e em uma GPU; levando em conta, claro, o tempo de execução dos mesmos.

1.3 Resultados esperados

No final do estudo, se alcançar-se os objetivos específicos mencionados anteriormente, ou seja, entender as GPUs e programar uma GPU com CUDA, então terá-sse um documento completo com conteúdo suficiente para um curso introdutório em programação de GPUs.

2. REFERENCIAL TEÓRICO

Neste capítulo damos uma volta completa em torno das GPUs. Na seção 2.1 é feita uma apresentação resumida das CPUs; Em seguida, a seção 2.2 é focada na arquitetura do sistema com GPU; a seção 2.3 trata da programação das GPUs; logo após, na seção 2.4 é apresentado o nascimento do termo GPGPU; e por fim, a seção 2.5 traz uma visão geral da plataforma CUDA.

Até a seção 2.4, este capítulo é baseado principalmente nos livros ***Organização e Projeto de Computadores, e Arquitetura de Computadores*** *de PATTERSON e HENNESSY, apenas parágrafos retirados em outras fontes sãos referenciados.*

2.1 GPUs

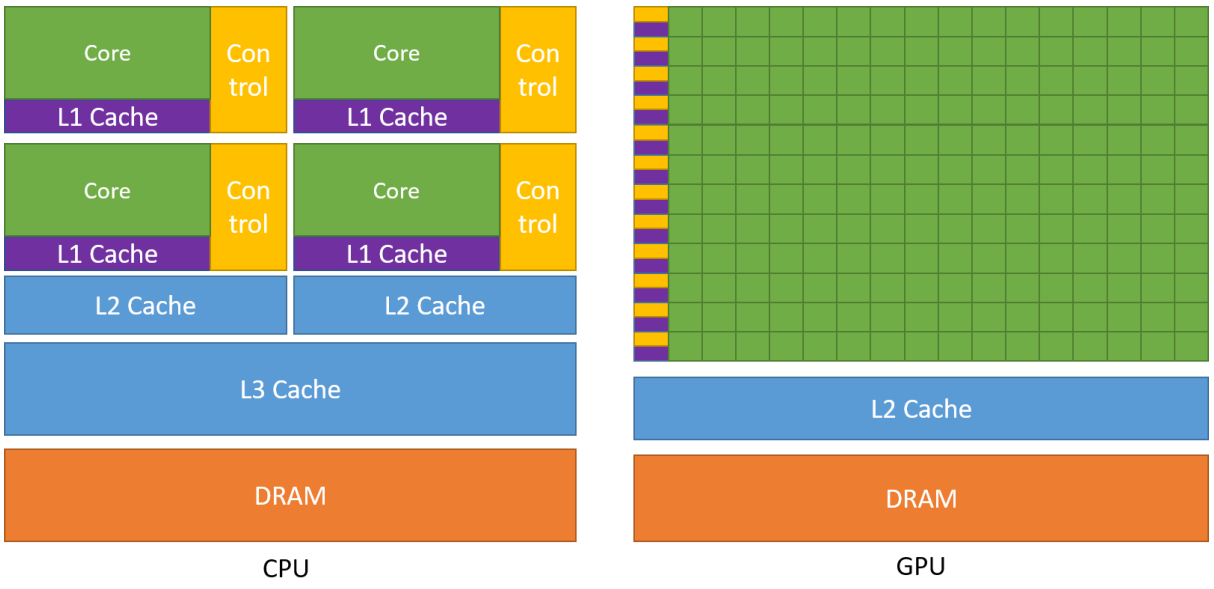
GPU do inglês *Graphics Processing Unit* (para Unidade de Processamento Gráficos) é um multiprocessador paralelo equipado de centenas de *cores* e milhares de threads, proporcionando um maior desempenho no processamento gráfico. Esta última pode ser incorporada em placas gráficas que, geralmente, usa a extensão Componente de Interconexão de Perifericos (PCI - *Peripheral Component Interconnect*) nos PCs ou em placa mãe de notebook, entre outros.

Uma GPU é um processador dedicado e otimizado para acelerar a computação gráfica nos computadores. Na essência, a GPU foi desenhada para computar pontos flutuantes para melhor performance na renderização 3D. As mais modernas são massivamente paralelas, programáveis e o poder de computação de pontos flutuantes encontrada nas mesmas é de longe superior à computação das CPU (McCLANAHAN, 2010).

As GPUs contam com uma taxa elevada de transferência de instruções e uma baixa latência em acesso de memória em comparação à CPU. Hoje em dia, muitos programas de aplicação aproveitam o alto desempenho dessas últimas, para serem executados mais rapidamente. A diferença na atuação de uma GPU em relação a uma

CPU existe porque, na essência, foram projetadas com propósitos diferentes: a CPU foi projetada para executar uma série de operações no menor tempo possível e tem o poder de processar dezenas de threads em paralelo; enquanto, a GPU foi projetada para se destacar, em executar milhares delas paralelamente. A GPU é construída, especialmente, para realizar cálculos excessivamente paralelos; logo, muito mais transistores são dedicados ao processamento de dados, em vez de usar caches e controle de fluxo. A distribuição de recursos (caches, transistores, etc.) para GPU e CPU é mostrada na figura 1. Na CPU tem-se cada core com sua grande cache L1 local e bastante transistores destinados ao controle de fluxo, conforme o tamanho da parte amarela, além das outras caches muito maiores, L2 e L3, disponíveis antes de enxergar a memória. Diferentemente, a GPU usa caches e controles de fluxo menores (observe as mesmas cores [amarelo, verde, etc.] das figuras) para um conjunto de cores (núcleos); em sua arquitetura é dispensada a cache L3, o que permite um acesso mais rápido a memória, e consequentemente, baixa latência (NVIDIA, 2021).

Figura 1: Distribuição de recursos ao processamento de dados em CPU e GPU



Fonte: NVIDIA, 2021

As GPUs conseguem uma baixa latência ao acessar a memória, pois não contam com grandes caches de dados e controle de fluxo complexo; entretanto, são ambas caras em questão de uso de transistores. Como, geralmente, os programas de

aplicação contêm partes sequenciais e partes paralelas, as arquiteturas combinam as duas entidades para obter maior desempenho, assim, programas altamente paralelos conseguem aproveitar a natureza massivamente paralela da GPU para acelerar a sua execução (NVIDIA, 2021).

2.1.1 Nascimento das GPUs

Tudo começou com os jogos, na década de 1990, com os fabricantes, como a Intel, cujos processadores não tinham grande poder de computação de pontos flutuantes, por exemplo o Intel 486. Os jogos nessa época, mesmo requerendo pouca gráfica em comparação aos de hoje, eram lentos; isso porque, a maioria dos processadores não tinha uma FPU (*Floating Point Unit*) embutida capaz de fazer operações de pontos flutuantes pesadas e de forma rápida. Uma possibilidade de melhoramento consistia em acrescentar uma unidade de processamento vetorial às CPUs, que eram capazes de processar várias operações de pontos flutuantes em paralelo. Embora essas unidades ajudassem certas aplicações, mas a demanda de energia era cada vez maior e os jogos exigiam sempre mais desempenho. Durante o período em que placas plug-in (som, ethernet, etc.) eram fabricadas, surgiu a ideia de criar placas que pudessem auxiliar no processamento de pontos flutuantes, e consequentemente, na renderização 2D e 3D. O esforço dos fabricantes em apresentar um produto ao mercado de jogos dá origem às Unidades de Processamento Gráfico, que após 1990 se tornaram parte indispensável dos computadores, não apenas para os jogos, mas também para outras afinidades (SOYOTA, 2018).

A evolução das GPUs veio de um controlador VGA (*Video Graphics Array*) com capacidade limitada para um processador paralelo programável. Em conjunto com a Unidade Central de Processamento, esses controladores de VGA, nos anos 90, incorporaram funções de aceleração tridimensional graças ao avanço significativo da tecnologia dos semicondutores. Conforme a evolução desses controladores, o termo GPU foi utilizado para indicar que o dispositivo gráfico passou a ser um processador, pois nesse último foram incorporados todos os detalhes do pipeline gráfico. Além

disso, o pipeline foi alterado, tornando os estágios do mesmo menos especializados e mais programáveis.

As GPUs revolucionaram a prática da computação gráfica e a visualização. Essa revolução teve como *booster* a indústria de jogos com sua constante demanda de eficiência no processamento 3D, as GPUs tiveram um drástico crescimento em performances e funcionalidades em poucos anos. Embora os hardwares gráficos foram desenhados, de início, para uma rápida renderização 3D, os mesmos podem ser usados para outros fins computacionais (WEISKOPF, 2007).

2.1.2 Finalidade das GPUs

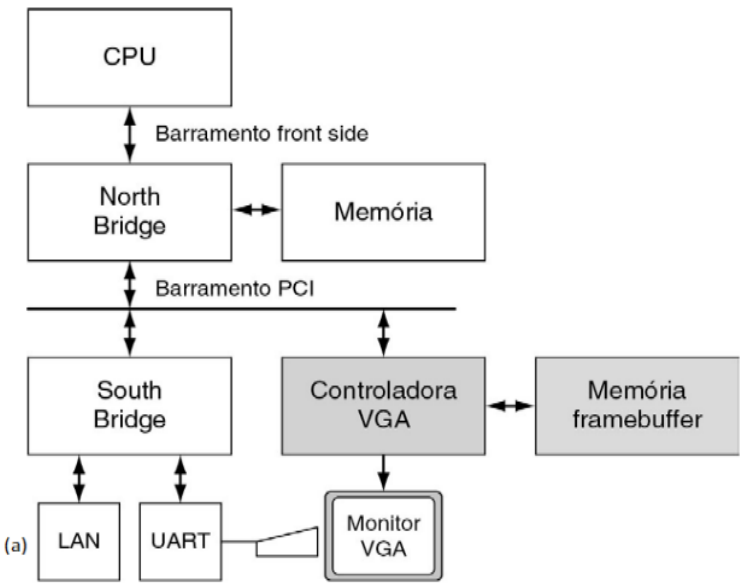
A principal tarefa das GPUs é o processamento de gráficos, permitindo aliviar a carga do processador, embora as mesmas possam ser programadas para executar outras tarefas. Com o tempo, na sua arquitetura, a lógica dedicada de função fixa é substituída, mas mantém-se a organização básica do pipeline gráfico 3D. Além disso, a precisão nos cálculos progrediu da aritmética indexada até pontos flutuantes de precisão dupla. As GPUs tornaram-se processadores programáveis com um denso paralelismo com milhares de threads e centenas de *cores* em que, recentemente, foram acrescentadas instruções de processador e hardware de memória dando suporte às linguagens de programação como C e C++, tornando a GPU um processador multicore programável de uso geral.

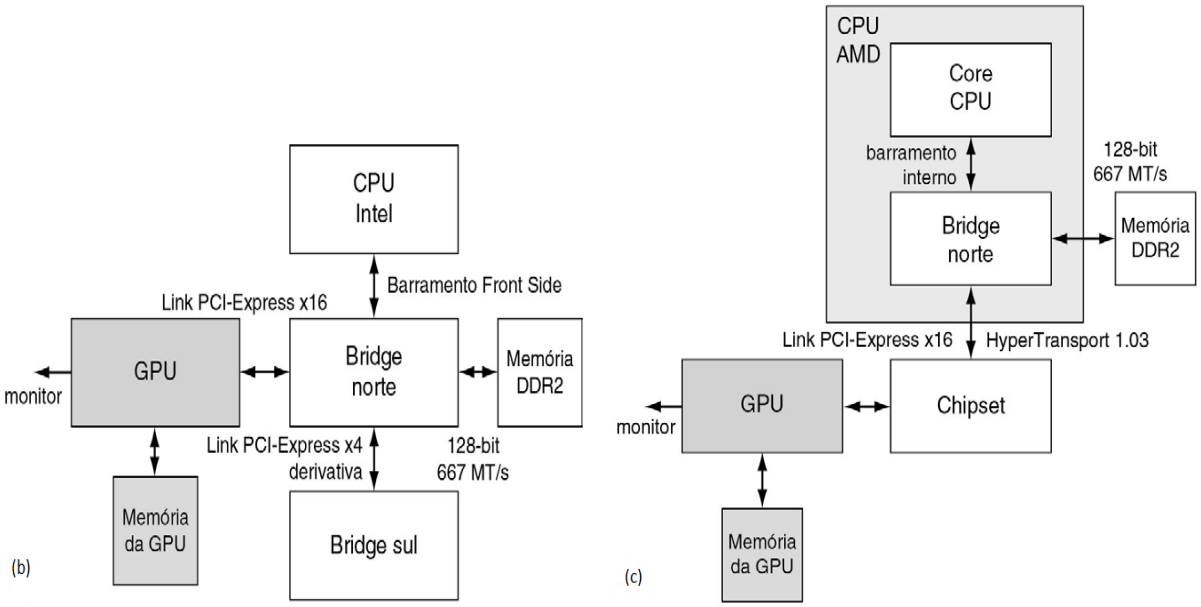
2.2 Arquitetura de sistema com GPU

Em um sistema de computador sem GPU (nos sistemas mais antigos) a exibição gráfica é controlada por um subsistema de *framebuffer* simples, conectada ao barramento PCI, pois o processador gráfico (GPU) ainda não existia. Hoje em dia, os sistemas de computadores são construídos com uma arquitetura heterogênea que envolve uma CPU e uma GPU separadamente. Nestes sistemas mais atuais, são

levados em conta características como o número de subsistemas funcionais e chips que são usados, tecnologia de interconexão, topologia, e também, memórias disponíveis aos subsistemas funcionais. São mostrados três sistemas de computadores na figura 2, sendo o primeiro um diagrama de contexto de um computador sem presença de uma GPU, ou seja, a exibição é controlada por uma placa VGA; no segundo um sistema com uma GPU e uma CPU Intel; e por fim, um esquema com uma CPU AMD e uma GPU. Na figura 2 (a), interfaces de alta largura de banda são usadas para interconectar a CPU, a memória e o barramento PCI na bridge norte; na bridge sul são utilizadas interfaces de conexão como o barramento ISA (para LAN, Áudio, etc.), controlador de interrupção, controladora de DMA, entre outros. No segundo caso (ou seja, figura 2 (b)), a GPU está conectada com o sistema usando uma interface PCI-Express 2.0; esta configuração permite uma taxa máxima de transferência de 16 GB/s, sendo 8 GB/s em cada direção. Na figura 2 (c), a mesma interface de conexão é usada para interconectar o chipset à GPU com as mesmas configurações. Vale ressaltar que, nesses dois últimos casos a GPU é discreta, isto é, há uma separação física entre a CPU e a GPU; embora os mesmos possam acessar a memória um do outro, obviamente, com uma latência maior do que a de acesso às memórias próprias. Uma outra observação, é que no sistema três (3), ou seja, o sistema com CPU AMD, a controladora de memória e a CPU fazem parte do mesmo subsistema.

Figura 2: Arquiteturas de sistemas de computadores sem e com GPU





Fonte: PATTERSON; HENNESSY, 2014

A arquitetura de hardware das GPUs evoluiu de um *unicore* específico e uma fixa implementação de pipeline destinada para processamento gráfico, para um conjunto de *cores* paralelos e programáveis de propósito de computação geral. Futuramente, GPUs terão muito mais cores paralelos e tornar-se-ão mais programáveis (McCLANAHAN, 2010).

2.3 Programação de GPUs

Nos dias atuais, falamos de *multicore* e *manycore* para caracterizar CPUs com mais de um *core* e GPUs com milhares de *cores*, em outras palavras, é dizer que os chips de processadores são, agora, sistemas de processadores paralelos; por exemplo, a GeForce RTX 3090 da NVIDIA lançada em 17 de setembro de 2020 equipada de 10496 *cores*. Enquanto o número de *cores* nas CPUs e GPUs aumentam, a maior dificuldade dos programadores é criar softwares auto adaptáveis para os processadores, isto é, softwares capazes de redimensionar-se, transparentemente, o seu nível de paralelismo conforme o aumento de *cores* nos mesmos; assim como os softwares gráficos 3D se redimensionam nas GPUs *manycore* equipadas de um número variável de núcleos (NVIDIA, 2021).

A programação de uma GPU é, em geral, diferente da programação de um processador *multicore*, pois as GPUs oferecem vários níveis de paralelismo de threads e dados a mais do que os processadores. O número de *cores* nas GPUs continua aumentando conforme a Lei de Moore, e para compensar desempenho e preço, nas GPUs, é alocado um número variável de *cores* e threads; pois, independentemente do número de processadores (*cores*) alocados, os usuários esperam sempre que seja suficiente para rodar aplicações gráficas, jogos, entre outros. Por isso, a forma de programar essas entidades e programas de aplicação que rodam nas mesmas, são projetados de maneira a se adaptarem transparentemente de acordo com o software. Modelos de programação como Cg (C para gráficos) e HLSL (*High-Level Shading Language*), e CUDA (veja a próxima seção) permitem que programas paralelos aproveitem um alto grau de paralelismo nos processadores e GPUs, respectivamente, expandindo de maneira transparente.

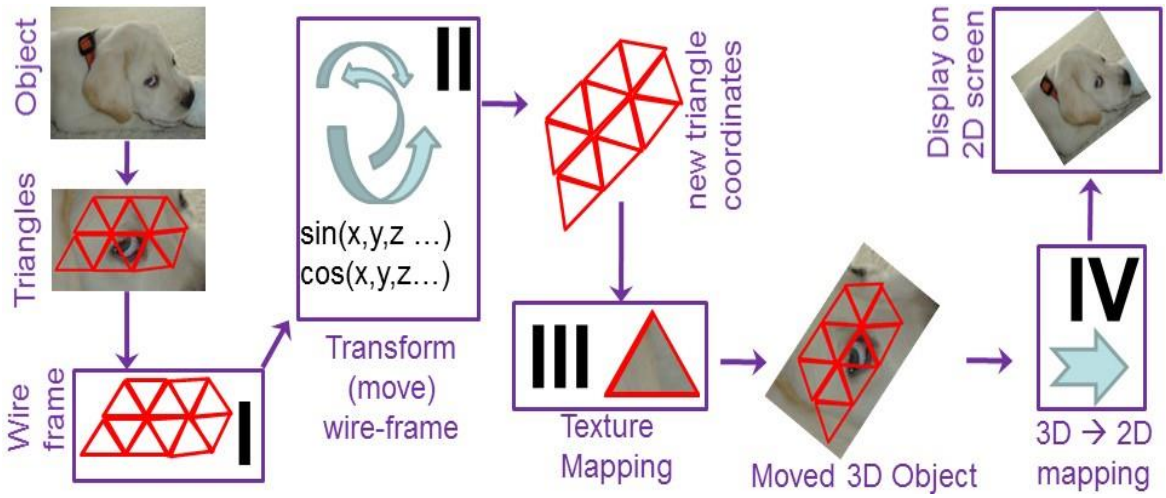
Nessas interfaces escaláveis de programação, o programador se preocupa apenas com uma thread, isto é, o mesmo escreve apenas o código para uma thread, e a GPUs executa milhares de instâncias em paralelo. Assim os programas se expandem, implicitamente, de acordo com o número de cores e threads disponíveis na mesma. Vale ressaltar que, duas interfaces padrões são usadas no desenvolvimento de programas gráficos em GPUs, são elas OpenGL e Direct3D; mas, para fins gerais são usadas as plataformas CUDA, Brook (linguagem de streaming) e CAL (*Compute Abstraction Layer*) na qual modelo de linguagem assembly. Em uma arquitetura amplamente paralela, o desenvolvedor ou o compilador mapeia grandes problemas, decompondo-os em muitas partes pequenas que possam ser processados de maneira concorrente. Nesse cenário, o multiprocessador paralelo computa blocos de resultados, e threads paralelas calculam elementos de resultados.

Como visto anteriormente, os programas de aplicação, em geral, contêm uma parte sequencial e uma parte paralela, e podem ser acelerados por GPUs. Nesses programas, a parte sequencial é executada na CPU, pois é aprimorado para um bom desempenho executando threads únicos; a parte paralela, por sua vez, é executada em milhares de núcleos e/ou threads paralelos na GPU (NVIDIA DEVELOPER, 2021).

2.4 General Purpose Graphic Processing Unit - GPGPU

Os primeiros fabricantes de GPUs perceberam que as mesmas podiam se destacar mais ainda se incorporado hardware dedicado a ela possibilitando, como mostrado na figura 3, operações como: operação sobre triângulo de dados (I), realizar operações pesadas de pontos flutuantes (II), associar textura aos triângulos e fazer uso da memória de textura (III), conversão de coordenadas de triangulares em coordenadas de objeto para exibição (IV). As GPUs foram evoluindo incorporando essas operações, tornando as mesmas mais poderosas. Embora fossem projetadas para o mercado de jogos, as GPUs passaram a ser usadas com outros propósitos, como em simulações de partículas físicas, simulação de circuito, biologia computacional, principalmente nas universidades e indústrias. Daí nasceu o termo GPGPU, para dizer que as GPUs se tornam processadores de propósito geral desviando-se da sua essência de processador gráfico. Nessa abordagem em GPGPU, os triângulos têm sua localização e sua textura como atributos. Detalhando as operações da figura 3, tem-se, primeiro, o objeto é mapeado em coordenados triangulares (I); depois, o mesmo é movido usando cálculos matemáticos sobre os seus coordenados (II); Em seguida, uma textura mapeada é aplicada sobre objeto movido (III); por fim, é executado um mapeamento 3D para 2D, para que a imagem possa ser exibida em um monitor convencional de computador (SOYOTA, 2018).

Figura 3: Operações para mover um objeto 3D triangulado.



Fonte: SOYOTA, 2018.

Os fabricantes perceberam grandes oportunidades, pois as universidades estavam comprando bastante as GPUs, então melhoraram essas últimas eliminando transformações tediosas, para facilitar o uso científico das mesmas. A NVIDIA, em particular, percebeu que era necessário facilitar os programadores de GPGPU, ou seja, os que programam aplicações para GPUs para executar tarefas que não sejam, especificamente, de processamento gráfico (jogos, etc.); então, julgou importante introduzir meios de programação para os desenvolvedores, sendo assim, no final de 2006 para o início de 2007 introduziu CUDA (SOYOTA, 2018).

2.5 CUDA

Esta seção é baseada da documentação de CUDA toolkit e o CUDA C++ Programming guide da NVIDA, logo, apenas partes retiradas de outras bibliografias são referenciadas.

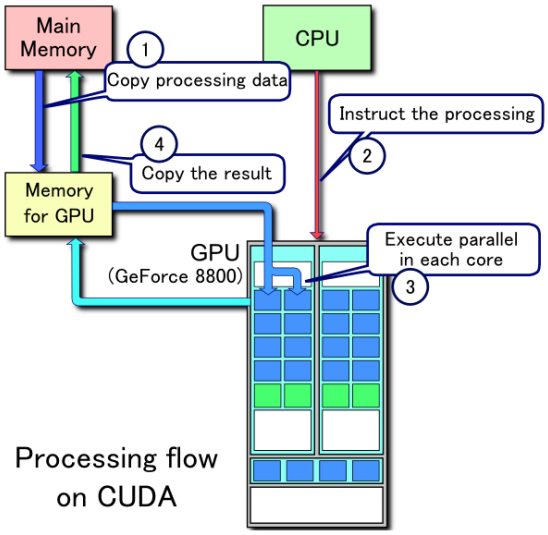
CUDA (*Compute Unified Device Architecture*), é uma interface de computação paralela e um paradigma de programação desenvolvido pela NVIDIA em novembro 2006, permitindo que programadores desenvolvam programas de propósito geral que possam ser executados na GPU, aproveitando o poder de computação da mesma para obter maior ganho de speedup. A CUDA pode ser usada com linguagens de programação como C/C++, Fortran, Python e MATLAB; com elas, o desenvolvedor especifica o paralelismo usando algumas palavras chaves simples. A interface conta com um pacote de ferramentas *(CUDA Toolkit*) disponibilizado, também, pela NVIDIA, contendo recursos suficientes para desenvolver aplicações aceleradas pela GPU; nele, são embutidos um compilador, bibliotecas de aceleração de GPUs, mecanismos de desenvolvimento e um ambiente de execução.

2.5.1 Arquitetura CUDA

Diferentemente de arquiteturas de computação gráfica anteriores que subdividem recursos computacionais em vértices e efeitos de pixel, a CUDA provê um pipeline de efeito unificado permitindo que Unidade de Aritmética Lógica (ALU) dos processadores processem instruções de programas de aplicação que não sejam de processamento gráfico. Isso porque, a NVIDIA projeta suas novas placas de vídeos de maneira a serem usadas para computar, não apenas instruções de renderização gráficas, mas também aplicações de uso geral. A unidade de processamento GPU é capaz de ler e escrever na memória, assim como em cache de aplicação de memória compartilhada. As características da arquitetura da CUDA são adicionadas no intuito de oferecer em uma GPU, além de processamento gráfico, possibilidades de um bom desempenho em aplicações de uso geral (SANDERS; KANDROT, 2010).

Na figura 4, é mostrado o fluxo de execução de um programa na arquitetura CUDA. Primeiramente, os dados que serão processados são copiados da memória da máquina host para a memória local da GPU; em seguida, a CPU faz a chamada do kernel, iniciando a sua execução na GPU; várias threads ou blocos de threads iniciam o processamento; e quando terminar, o retorno é copiado para a memória host (*Main memory*) para que o processador tenha acesso ao mesmo (MALANÍK, 2010).

Figura 4: Fluxo de processamento na arquitetura CUDA



Fonte: MALANÍK, 2010

A arquitetura conta com um conjunto de palavras-chaves específicas à mesma, a fim de facilitar a programação e alcançar o maior número de programadores possíveis. Alguns meses depois do lançamento de GeForce 8800, a NVIDIA lançou um compilador para a linguagem CUDA C, e a mesma se tornou a linguagem de programação especializada em programar GPUs para programas de uso geral (SANDERS; KANDROT, 2010).

2.5.2 Programação escalável

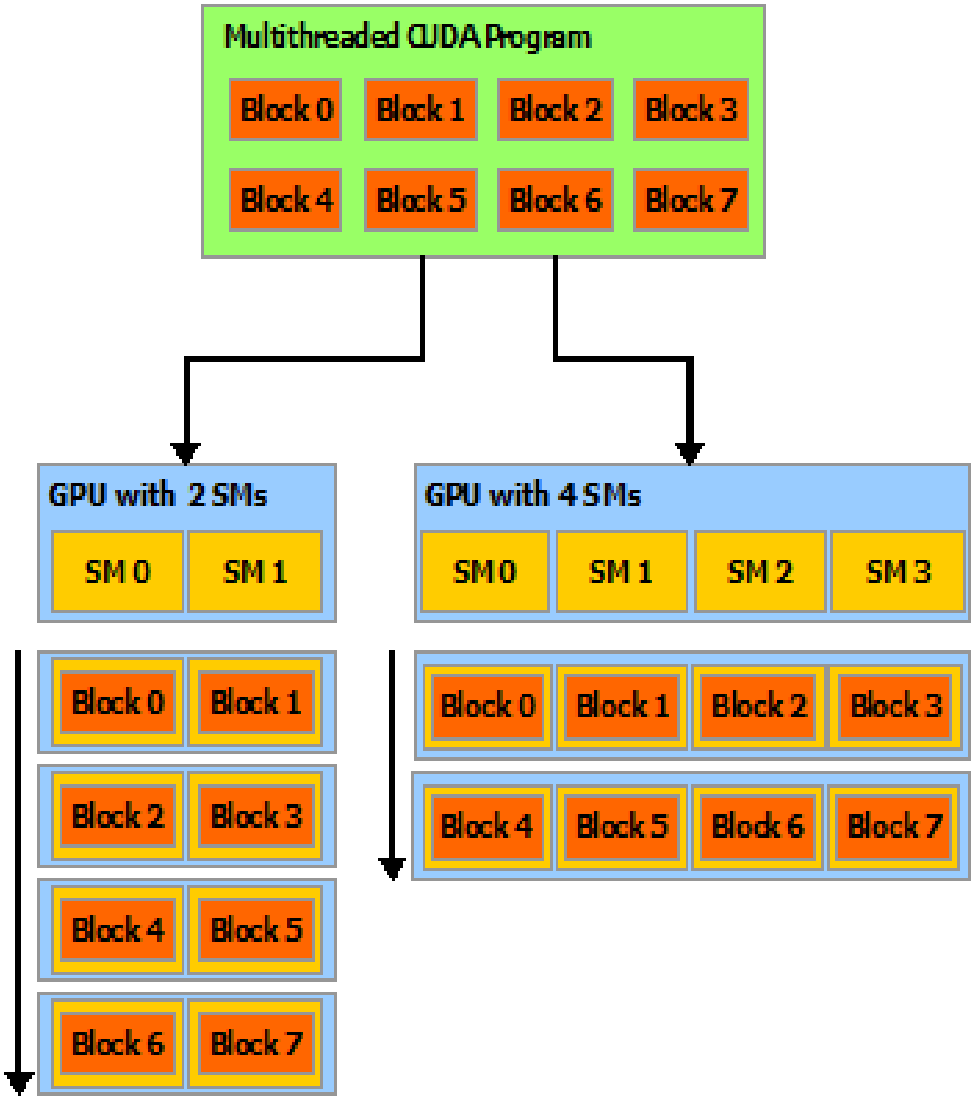
O paradigma de programação CUDA foi projetado para avantajar-se os programadores, do desafio que representa o desenvolvimento de programas paralelos auto adaptáveis; requerendo uma baixa taxa de aprendizagem dos que já tiveram conhecimento em linguagens de programação, como C. O modelo contém em sua base três níveis de abstração visíveis ao programador como ampliações de linguagem, são eles: uma organização hierárquica de grupo de threads, memórias compartilhadas e uma sincronização de barreira.

Essas abstrações oferecem um paralelismo de dados de granularidade baixa e paralelismo de threads, pertencentes ao paralelismo de dados de granulação grossa e paralelismo de tarefas. Assim, o programador pode decompor o problema em partes independentes que possam ser computadas paralelamente por blocos de threads; e cada parte destas, pode, também, ser particionada gerando partes menores que podem ser processadas solidariamente em paralelo por todas as threads de um determinado bloco. Vale notar que, a decomposição do problema preserva a expressão da linguagem, permitindo que threads cooperem entre si na resolução de cada subproblema, e ainda mantendo a escalabilidade. Cada bloco de threads pode ser programado em qualquer multiprocessador da GPU, sequencialmente ou paralelamente, faz com que um programa CUDA compilado possa ser executado em qualquer número de multiprocessador.

Nesta abordagem, apenas o ambiente de execução CUDA precisa ter conhecimento do número de processadores disponíveis. Um programa multithreads é

subdividido em blocos de threads processados de maneira independente, permitindo que uma GPU com mais multiprocessadores, automaticamente, compute o mesmo mais rapidamente do que uma com menos multiprocessadores. A figura 5 mostra a auto-escalabilidade de um programa multithreads. O mesmo programa em uma GPU com 2 SMs usa 8 blocos sendo 4 de cada SM, em uma outra arquitetura com 4 SMs, o mesmo se auto-escala usando apenas dois blocos de cada SMs.

Figura 5: Escalabilidade automática de um programa multithreads CUDA



Fonte: NVIDIA - CUDA C++ Programming Guide, 2021.

2.5.3 Modelo de programação

No modelo de programação CUDA C++, os desenvolvedores podem definir funções C++, fazer chamadas de kernels, entre outros, pois este estende C++. Essas funções, quando disparadas, são executadas N vezes em paralelo por N threads CUDA, diferentemente de uma chamada regular em C++. Neste modelo, uma sintaxe particular é usada para definir e configurar o kernel. Observe o exemplo a seguir, é um código resumido para a adição de dois vetores A e B que, depois, armazena o resultado no vetor C.

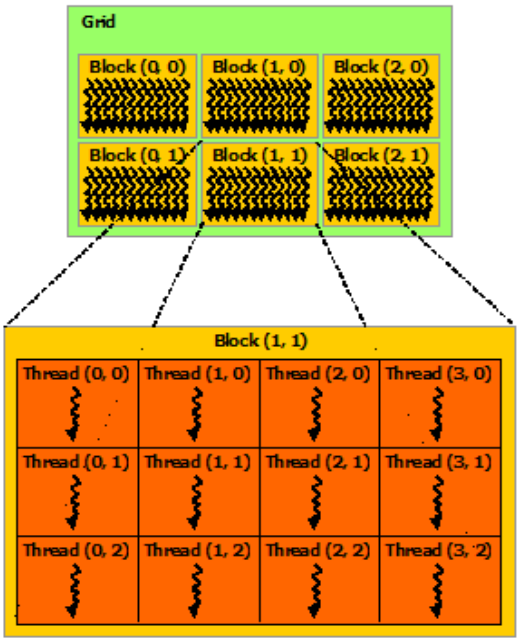
```
//definição do kernel
__global__ addVector(float *A, float *B, float *C, int size)
{
    int threadId = (blockDim.x * blockIdx.x) + threadIdx.x;

    if(threadId < size)
        C[threadId] = A[threadId] + B[threadId];
}

int main()
{
    //cirar N
    //criar e preencher A e B
    //criar C

    //chamada do kernel
    addVector<<<1, N>>>(A, B, C, N);

    return 0;
}
```

Neste exemplo, o especificador de declaração `__global__` é usado para definir o kernel, o restante do corpo da função é comum em linguagem C/C++. A configuração de execução do kernel é feita usando <<< … >>>, dentro é especificado o número de blocos de threads e o número de threads CUDA por bloco a serem usados na computação desse kernel em uma invocação específica. Um identificador único de encadeamento é atribuído a cada cadeia que processa o kernel, e esse pode ser acessado no mesmo mediante variáveis integradas. Vale notar que, cada thread do conjunto N que computa o kernel do exemplo, faz uma adição de pares.

O `threadIdx`, por padrão, é um vetor de três elementos, permitindo que threads possam ser identificadas com apenas um ou dois, ou mesmo três índices; e conforme o número de índice, forma blocos de threads uni, ou bi, ou tridimensional. Essa forma de identificação facilita a especificação de elementos de um vetor (geométrico), indexação de elementos de matrizes, entre outros. É desejável que as threads de um determinado bloco residam em um mesmo *core* da GPU, fazendo com que exista um limite para o número de threads em um bloco; essas últimas, também, devem limitar-se ao compartilhamento dos recursos exclusivos a esse núcleo. Atualmente, esse número pode chegar até 1024 threads em um bloco de uma GPU. Contudo, vários blocos de threads podem executar o mesmo kernel, logo, o número total de threads participantes nessa computação é um produto do número de blocos de threads executados vezes o número de threads por bloco.

O número de threads por bloco e o número de bloco por grade especificado no `<<< … >>>` pode ser de tipo inteiro ou dim3 sobre qual falaremos mais adiante. Semelhantemente às threads, um bloco em uma grade contém um identificador único e pode ser acessado, dentro do kernel, por indexação uni, bi, ou tridimensional mediante variável embutida como `blockIdx`; e o tamanho (dimensão) do bloco pode ser acessado via a `blockDim`, também acessível dentro do kernel. Cada bloco de threads deve ser executado independentemente dos outros, e os elementos do conjunto (os blocos) devem ter uma execução sem ordem exclusiva, pode ser em série ou paralelo. Essa independência requerida possibilita que blocos de threads possam ser processados em qualquer ordem em qualquer número de núcleos, e isso faz com que, programadores escrevam códigos escaláveis com o número de núcleos. Conforme a figura 6, a organização dos blocos de threads é feita usando uma grade uni, bi ou tridimensional. O número de blocos de thread em uma grade é definido pelo tamanho dos dados que estão sendo processados, o que geralmente, excede o número de processadores no sistema.

Figura 6: Mostra a identificação dos blocos dentro de uma grade e das threads dentro de um bloco.



Fonte: NVIDIA, 2021.

As threads dentro de um bloco podem cooperar entre si, compartilhando dados em uma memória compartilhada e sincronizando a execução das mesmas para administrar o acesso à memória; precisamente, uma pode especificar um ponto de sincronização no kernel pela chamada da função `__syncthreads()`. Esta última atua como uma bolha (*stall*) em pipeline, isto é, garantir que threads que terminam primeiro seu processamento, parem e esperem para os demais, antes de passar para a próxima etapa. A memória compartilhada neste caso, deve ser uma memória com baixa latência e próxima aos núcleos do processador (como Cache L1), proporcionando uma eficiente cooperação, e a `__syncthreads()` uma função com menor complexidade de tempo e espaço possível.

2.5.4 CUDA Toolkit

O *CUDA Development Toolkit* contém o conjunto de recursos necessários para desenvolver na plataforma, pois o mesmo conta com tecnologias, APIs e ferramentas

que facilitam o desenvolvimento com CUDA. Neste último, são disponibilizados recursos para distribuir tarefas em configurações multi-GPUs, isto é, estrutura com várias GPUs, permitindo que cientistas e pesquisadores desenvolvam aplicações de se auto-dimensionam em estações de trabalho, assim como em arquitetura com milhares de GPUs. Além disso, é incorporado o compilador CUDA C/C++, que serve para compilar e executar os algoritmos que são implementados durante este estudo.

Em algumas das APIs disponíveis no kit de ferramentas CUDA, alguns cabeçalhos interessantes de se olhar são, entre outros. `<cuda_runtime.h>` e `<cuda_runtime_api.h>`, `cuda_runtime` é um cabeçalho C++ enquanto `cuda_runtime_api` é um cabeçalho feito em C que contém funções host, ou seja, funções que devem ser chamadas apenas na máquina host, e declaração de tipos. Por exemplo, `cudaFree(*pointer)` que desaloca uma estrutura de dados alocada com `cudaMalloc(....)`. Vale ressalatar que `cuda_runtime_api` é um subconjunto de `cuda_runtime` e que estes são partes principais da API Runtime. O cabeçalho `<cuda.h>` é usado para referenciar a API Driver, o mesmo possui funções que permite ao programador manter o controle sobre a execução do programa, por exemplo,

```
CUresult cuFuncGetAttribute ( int* pi, CUfunction_attribute attrib, CUfunction hfunc )
```

que permite o tracking de atributos de uma função em execução, `pi` é a variável de retorno do valor do atributo, `attrib` é o atributo a ser rastreado, e `hfunc` a função a ser controlada. `<thrust/.......>` a API Thrust possui métodos semelhantes à Biblioteca Padrão de Template (*STL – Standard Template librairy*) de C++, no próximo capítulo é apresentada com mais detalhes.

3. PRIMEIRO PASSO COM CUDA

Neste capítulo é feita uma introdução a CUDA na qual são apresentados, de maneira resumida, alguns tópicos como: criar, compilar e executar um programa CUDA; O programa hello-world; apresentação das APIs usadas nesse estudo; alguns conceitos básicos de CUDA e análise de exemplos usando as APIs runtime e thrust, sobre os quais falaremos posteriormente.

A bibliografia usada neste capítulo é a documentação do CUDA Toolkit, especificamente, das APIs Thrust, Runtime e Driver. Portanto, somente segmentos retirados em outras bibliografias são referenciados.

3.1 Criar, Compilar e Executar um programa CUDA

A CUDA usa a extensão `.cu` para os arquivos de código fonte. Depois de criar o arquivo, é preciso escolher uma (ou mais) API CUDA para o desenvolvimento, conforme o objetivo. A compilação de tal programa usa o compilador da NVIDIA `nvcc` para qual, em todas as plataformas, podem ser usados os comandos abaixo para compilar e executar, respectivamente, um código feito em C/C++ (FARBER, 2011):

```
$ nvcc nome-do-programa.cu -o nome-do-objeto
$ ./nome-do-objeto
```

Vale ressaltar que, para programa contendo kernel recursivo, é preciso uma configuração a mais. Veja apêndice B.

3.2 Primeiro programa CUDA

Como é feito, geralmente, aqui é implementado um programa “Hello world” para ser executado por 4 threads CUDA. Primeiro, é criado um kernel para imprimir “**Hello**

world!”. Inicialmente, crie uma pasta `hello-world` na área de trabalho (Desktop), dentro dela, crie um arquivo `hello-world.cu`, e nele, colocar o código a seguir:

```
//hello-world.cu
//kernel, esta função é executada na GPU
 __global__ void helloWorld(){
    printf("\n Hello world!\n");
}

 int main(){

    helloWorld<<<1, 4>>>();

    return 0;
 }
```

Agora, para compilar o programa, abrir um terminal usando `Ctrl + Alt + T`; Use o comando `$ cd ~/Desktop/hello-world`, para entrar na pasta; execute o comando `$ nvcc hello-world.cu -o hello-world` para compilar o programa; e em seguida, use o comando `./hello-world` para executá-lo; e percebe, no terminal, *“hello world!”* impresso 4 vezes. Note que, para esse programa, não foi preciso incluir nenhuma biblioteca específica de CUDA Toolkit, mas pode-se fazer isso a vontade, como por exemplo, o cabeçalho `<cuda.h>`.

3.3 Apresentação das APIs

Bastante APIs são disponibilizadas para a programação utilizando CUDA, por exemplo, a API **Thrust** que trabalha com paralelismo de dados para C++; a **Runtime** usada tanto em C quanto em C++; a **Driver**; entre outras. A escolha de uma ou mais para se trabalhar, depende do nível de controle que o programador deseja ter sobre a execução do seu programa na GPU; quanto mais baixo é o nível da API, mais controle se terá sobre a execução do programa. Neste caso, tem-se os níveis na ordem decrescente da Thrust > Runtime > Driver. A seguir é, resumidamente, apresentadas essas últimas, que são usadas nas implementações dos kernels neste estudo.

3.3.1 API Thrust

A Thrust API é baseada na Biblioteca de Template Padrão (*Standard Template Library - STL*) de C++. A Thrust facilita os programadores a escrever aplicações paralelas usando interfaces de alto nível. Essa última é rica em primitivos paralelos de dados como métodos para somar, ordenar, varrer uma lista, e muito mais. A combinação desses métodos permite a implementação de algoritmos complexos sem muito esforço e gasto de tempo, com código limpo e intuitivo. A API Thrust conta com alguns cabeçalhos como `<thrust/device_vector.h>`, `<thrust/host_vector.h>`, `<thrust/reduce.h>`, `<thrust/sequence.h>`, etc. que contêm mecanismo para alocar um vetor na memória da GPU, reservar um vetor na memória da máquina host, reduzir os elementos de uma lista, gerar uma lista de elementos seguindo uma sequência, respectivamente. Vale ressaltar que, esses tipos de containers permitem armazenar qualquer tipo de dados e podem ser redimensionados dinamicamente.

As APIs de alto nível, como a Thrust, é largamente usada para prover uma alta performance computacional; contudo, o programador não tem grande controle sobre o seu trabalho, pois a mesma, consegue tomar decisões próprias durante a execução do programa. Essas APIs facilitam o programador, pois esse último consegue escrever códigos claros e sustentáveis rapidamente. A principal desvantagem das APIs de alto nível, como a Thrust, é o isolamento do desenvolvedor em relação a GPU, expondo apenas uma parte dos poderes da mesma ao programador. No entanto, com o uso da sintaxe da linguagem C++, para um programador científico, essas APIs representam uma boa saída. Portanto, enquanto não for necessário um controle maior sobre o hardware, pode-se usar, tranquilamente, as APIs de alto nível (FARBER, 2011). Observe o código abaixo que mostra um exemplo de alocação com a Thrust

```
//alocar um vetor de inteiro de 16 posições na memória da GPU
thrust::device_vector<int> v(16);
//alocar um vetor de inteiro de 16 posições na memória da GPU
//e preencher o mesmo com 0
thrust::device_vector<int> v(16, 0);

//alocar um vetor de carácter de 16 posições na memória da do computador
thrust::host_vector<char> v(16);
```

```
//alocar um vetor de carácter de 16 posições na memória da computador
//e inicializar as posições do mesmo com 'a'
thrust::host_vector<char> v(16, 'a');
```

Um cuidado a ser tomado, é que quando solicitar-se um `device_vector`, a Thrust aloca muito mais do que o tamanho solicitado; ou seja, quando se trabalhar com memória limitada, deve-se dar uma atenção a mais as alocações.

3.3.2 API Runtime e Driver

Não se deve confundir a API Driver com a Runtime apesar da semelhança entre as duas. A Runtime possibilita o gerenciamento do código com facilidade, gerenciamento de contexto e módulo simplificando o código fonte, mas não há o nível de controle que a API Driver possui. Embora o uso das mesmas seja intercambiável, há algumas diferenças importantes entre elas. Sente-se livre para trabalhar com APIs de baixo nível caso esteja buscando uma performance específica e um controle maior sobre a codificação. A API Driver, de baixo nível, permite ao desenvolvedor ter mais controle sobre a implementação, entretanto, é mais trabalhoso se usa, demanda mais esforço e mais linhas de códigos para uma mesma tarefa. Note que, a Driver não é o foco neste estudo, mas sim a API Thrust e Runtime.

A Runtime API é projetada de maneira a fornecer ao programador o acesso a todas as características programáveis da GPGPU. Usando a sintaxe da linguagem C, a Runtime possibilita o desenvolvedor a escrever códigos simples, claros e intuitivos, de grande manutenibilidade, além de serem extremamente eficientes. Uma vantagem das APIs de baixo nível é que o programador pode ter um controle preciso sobre a execução de seu programa. E, a principal desvantagem é o número alto de linhas de códigos, especificação de parâmetros nas várias chamadas de APIs feitas; além da necessidade de verificação de compatibilidade, e tratamento de erros em tempo de execução (FARBER, 2011). A seguir o exemplo anterior de alocação usando API Runtime.

```
//alocar um vetor de inteiro de 16 posições na memória da GPU
//e preencher a suas posições com 0
   cudaMalloc(&deviceVector, sizeof(int) * 16);
   cudaMemset(&deviceVector, 0, sizeof(int) * 16);

//alocar um vetor de inteiro de 16 posições na memória do computador
//e inicializar as posições do mesmo com 0
   int *hostVector = (int *) calloc(16, sizeof(int));
```

Uma curiosidade, é que, em geral, as funções e tipos da API Runtime começam com prefixo `cuda`, enquanto os da API driver começam com `cu`.

3.4 Conceitos básicos de CUDA

Já vimos exemplos de código utilizando a API Thrust, Runtime e Driver. Agora, para entender os exemplos apresentados na seção subsequente, são apresentados aqui, alguns conceitos que ajudarão no entendimento.

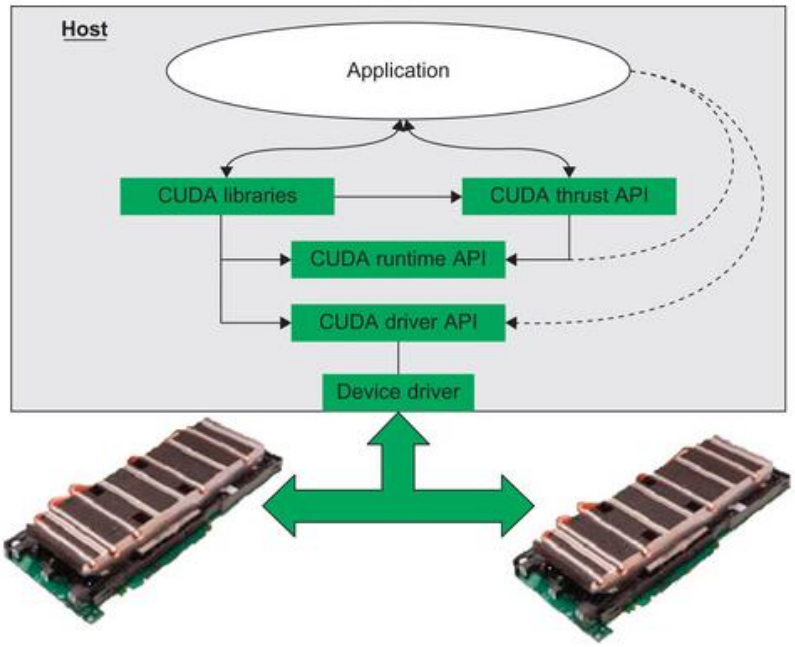
- **GPUs para CUDA,** são dispositivos separados conectados a um computador host. Pode-se conectar duas até quatro GPUs a um sistema, depende da capacidade de conexão do mesmo; por exemplo, número de porta PCIe, potência da fonte, e refrigeração. Cada GPU, para CUDA, é um dispositivo separado que trabalha assincronamente com o(s) processador(es) host, isto é, o(s) processador(es) host e as GPUs trabalham simultaneamente em conjunto. Transferência de dados ou instruções entre host e GPUs, ou até mesmo entre as placas, é feita usando as portas PCIe do sistema host. Várias formas de transferências são disponibilizadas por CUDA, entre essas temos (FARBER, 2011):
    - **Transferência explícita**, usando a API Runtime é feita através da chamada da função `cudaMemcpy()`, essa rotina requer quatro parâmetros, sendo o destino, a origem, o tamanho total da cópia, e o tipo da transferência. Veja abaixo o protótipo dessa última. Uma abordagem de alto nível, usando a Thrust, pode ser de uma forma mais simples. Observe o código a seguir:

```
1   //usando RUNTIME
2   //exemplo
3 cudaMemcpy(d_A, h_A, N * sizeof(int),
                        cudaMemcpyHostToDevice);
4   //usando THRUST, mais alto nível
5   d_A = h_A;
6   h_A = d_A;
```

Na linha 3, é feita a cópia de um vetor `h_A` de N inteiros da máquina host para `d_A` na GPU; linha 4, faz a mesma coisa, só que em alto nível; a linha 6, faz o inverso, da GPU para o computador host.

- **Transferência implícita,** nessa abordagem não é preciso a intervenção do programador, uma região da memória da máquina é mapeada para uma região da memória da GPU, assim, não é preciso operações de cópia para que a GPU acesse aos dados, e vice-versa. Portanto, economiza energia e custo de transferência, e consequentemente é muito eficiente. Essa forma de trabalhar é usada nas GPUs com consumo mínimo de energia. A figura 7, mostra que a aplicação e as APIs CUDA residem na máquina host, via PCI é conectada a GPU, com qual o software que faz uso das APIs comunica mediante driver.

Figura 7: Componentes da pilha de software CUDA



Fonte: FARBER, 2011

- **Memória separada da máquina host**, todas as GPUs possuem sua memória interna (exemplo, *RAM*), pois foram projetadas para prover uma alta performance computacional. Vale ressaltar que, há um conjunto mínimo de dispositivos que fogem desta regra. CUDA, a partir da sua versão 4.0, integrou endereços de memória virtual unificados (*UVA - Unified Virtual Addressing*) que permite que um programa em execução possa acessar a memória de um outro dispositivo usando apenas um ponteiro.
- **Uso de kernel**, nas rotinas executadas na GPU que são chamadas pelo sistema host. Um kernel, não é uma função, logo, não pode retornar nenhum valor. O uso do kernel é essencial a um bom desempenho da GPU, sua declaração é feita utilizando a palavra-chave `__global__`, isso informa ao compilador que este é chamado pelo sistema host.
- **Chamada assíncrona do kernel**, após a chamada do kernel, o processador host não para esperando que termine sua execução, ele continua executando outras tarefas. Para manter a GPU ocupada o máximo possível, uma pipeline pode ser criada com uma fila de kernels a serem executados. Pois, são chamados assincronamente, então é preciso uma sincronização nesta abordagem, por isso, é chamada a rotina `cudaDeviceSynchronize()` que faz com que o processador host espere até seja completa a execução de todos os kernels.
- **Thread**, é a unidade básica de processamento em uma GPU. Cada uma trabalha como se tivesse seu próprio processador, registradores e identidade. Um kernel pode usar N threads ou/e blocos de threads, essas informações são passadas na chamada do kernel entre os `<<< >>>` como na chamada de helloWorld(...).

```
helloWorld<<<nBlocks, nThreadsPerBlock>>>();
```

- **Memória global**, assim é chamada a maior região de memória compartilhada da GPU. Algumas regras são aplicadas sobre essa região para controlar as transações das threads na mesma, principalmente, as operações de leitura e escrita (*load e store*).

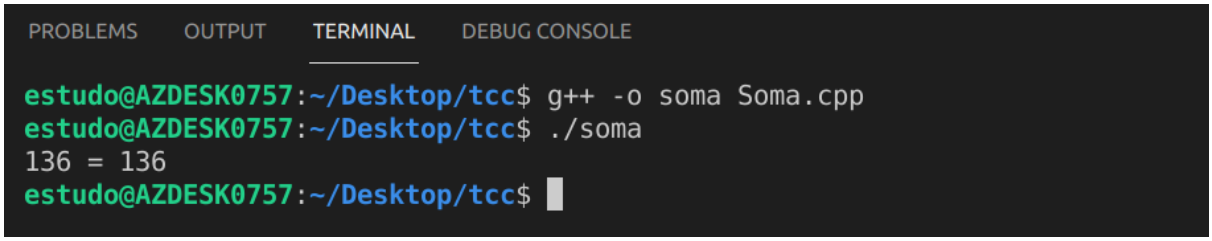
3.5 Analisando alguns exemplos

A seguir, um exemplo de um programa C++ que soma os elementos de um vetor.

```cpp
#include<iostream>
using namespace std;

int main()
{
   int vector[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
   int n = 16;
   int sum = 0, checkSum = 0;

   //soma dos elementos do vetor
   for(int i = 0; i < n; i++)
       sum += vector[i];

   //soma de verificação
   for(int i = 1; i <= n; i++)
       checkSum += i;

   //imprime os valores
   cout << checkSum << " = " << sum << endl;

   return 0;
}
```

O código sequencial C++ conta com dois (2) laços for, sendo o primeiro faz a soma dos valores e armazená-la na variável soma; e o segundo computar a soma dos índices, para verificar que a soma (variável) tenha o valor esperado. E a execução desse programa deu como saída mostrada na figura 8:

Figura 8: Mostrando o resultado da execução do código C++



Essa soma gasta tempo $O(n)$, pois é processado sequencialmente, ou seja, o processo soma elemento por elemento. Agora veja um exemplo da mesma soma usando a API Thrust de CUDA.

```
#include<iostream>
#include <thrust/reduce.h>
#include <thrust/sequence.h>
#include <thrust/device_vector.h>

using namespace std;

int main()
{
   int n = 16;
   //alocar um vetor de 16 posição na GPGPU
   thrust::device_vector<int> vector(n);
   //preenche o vetor
   thrust::sequence(vector.begin(), vector.end(), 1, 1);
   //soma dos elementos do vetor
   int sum = thrust::reduce(vector.begin(), vector.end(), 0);

   //imprime a soma
   cout << sum << endl;

   return 0;
}
```

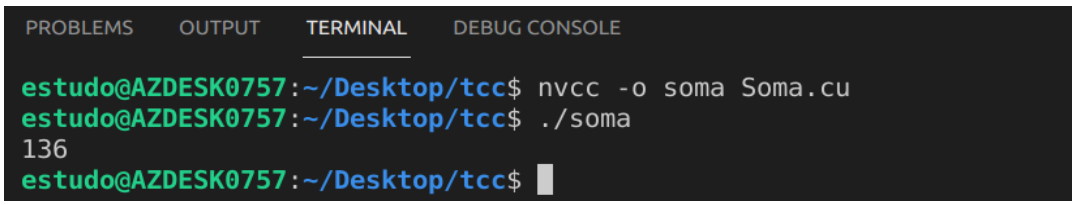
Usando a API **thrust** de paralelismo de dados de CUDA, o programa processa o mesmo objetivo do programa sequencial C++, ou seja, soma os valores do vetor. A diferença entre os códigos, é que, a somatória dos valores é feita em paralelo, usando várias threads CUDA. Observe que essa soma é calculada chamando a função `reduce` da API thrust que faz a computação em paralelo na GPU. Veja a seguir um outro exemplo com CUDA, desta vez, usando a API Runtime e Driver.

```
#include<cmath>
#include<cuda_runtime.h>

//Kernel Runtime, soma os elementos do vetor
__global__ void sumKernel(int *vector, int nElements, int nSteps)
{
   int threadId = (blockIdx.x * blockDim.x) + threadIdx.x;
   int nThreadToProcess = blockDim.x;

   while(nSteps > 0)
```

```
    {
        if(threadId < nThreadToProcess)
            vector[threadId] += vector[(threadId + nThreadToProcess)];

        nThreadToProcess = nThreadToProcess / 2;
        nSteps -= 1;
       //sincronização das threads
        __syncthreads();
    }
}

using namespace std;

int main()
{
    int nElements = 16;

    // a soma é feita em O(log n)
    int nSteps = (int) floor(log2(nElements));
    //cria o vetor (API Thrust)
    thrust::device_vector<int> vector(nElements);
    //preenche o vetor (API Thrust)
    thrust::sequence(vector.begin(), vector.end(), 1, 1);

    //soma dos elementos do vetor (chamando o kernel runtime)
    sumKernel<<<1, nElements/2>>>
                (thrust::raw_pointer_cast(&vector[0]), nElements,
                                                        nSteps);

    //imprime a soma
    cout << vector[0] << endl;

    return 0;
}
```

Note que, em comparação a implementação com API thrust, foi incluída a biblioteca de matemática de C++ e o cabeçalho `<cuda_runtime.h>`. O processo da soma é feito em tempo $O(log\ n)$, isso porque foi usado um esquema de árvore binária; logo, a complexidade é a altura da árvore, já que em cada subida na árvore a somatória é computada em paralelo, isto é, em tempo constante de $O(1)$. A execução das duas implementações gera a saída mostrada a seguir.

Figura 9: mostrando a saída para o programa da soma com Thrust e Runtime



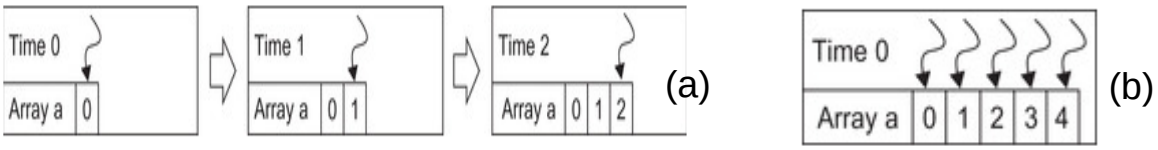
Pode-se observar a diferença em relação ao número de linhas de códigos da implementação usando a Thrust API e a Runtime API. A chamada de `thrust::reduce()` é substituída pela chamada de `sumKernel(...)` que é construída usando a API runtime. Percebe que, como a alocação do vetor é feita usando a API Thrust, uma chamada a `thrust::raw_pointer_cast()` foi preciso para que a API runtime possa processar os dados corretamente, tendo a localização atual dos mesmos na GPGPU. Na implementação do kernel, cada thread acessa duas posições no vetor e soma o conteúdo da posição mais à direita com o da posição mais à esquerda. Em cada subida na árvore, o número de threads assim como o número de índices válidos do vetor diminui pela metade, até chegar a apenas uma threads que reduz as duas primeiras posições do vetor em uma, sendo `vector[0]` com o valor final da soma. E mais uma curiosidade, é que a alocação e o preenchimento do vetor são feitos usando a API Thrust, apenas a soma é processada com a Runtime. Em uma versão, completamente, de baixo nível, teria as seguintes instruções para alocação e preenchimento.

```
//......
//inclusões #include<....>
//criação do kernel e outras operações...
//........
int main(){

   int *deviceVector = NULL;
   //alocação de um vetor na máquina host
   int *hostVector = (int *) malloc(sizeof(int) * nElements);
   //alocação do vetor na GPGPU
   cudaMalloc(&deviceVector, sizeof(int) * nElements);

   //preencimento do vetor na memória host
   for(int i = 0; i < nElements; i++)
       hostVector[i] = i;
```

```
   //copia o conteúdo do vetor hostVector da memória da máquina para o
deviceVetor na GPU
   cudaMemcpy(deviceVector,   hostVector,   sizeof(int)   *   nElements,
cudaMemcpyHostToDevice);

   //......
   //chama o kernel para processar a soma, passando o deviceVector
   //........

   //copia de voltar o vetor para máquina host
   cudaMemcpy(hostVector,   deviceVector,   sizeof(int)   *   nElements,
cudaMemcpyDeviceToHost);

   //..........
   //imprime o resultado e outras operações
   //................

   cudaFree(deviceVector);
   free(hostVector)

   return 0;
}
```

Observe a figura 10, (a) apresenta o modo de acesso usando apenas uma threads para o processo sequencial e (b) mostra o acesso usando uma thread por índice, isto é, processamento paralelo (FARBER, 2011).

Figura 10: Mostra a diferença na execução sequencial e paralela



Fonte: FARBER, 2011

3.6 Analisando o kernel runtime do programa da seção anterior

Analisando o programa a partir da rotina `main()`. A variável `nElements` define o número de elementos a ser alocado; a variável `nSteps`, por sua vez, define o número de etapas necessário para computar a soma; A chamada do método `thrust::device_vector<int>vector(nElements)` aloca o vetor vector com tamanho `nElements` na memória da GPU. Na chamada do método `thrust::sequence(vector.begin(), vector.end(), 1, 1)` é preenchido o conjunto com valores incrementados a partir de 1, ou seja, 1, 2, 3, …., 16; em seguida, é chamado o kernel que faz o processamento da somatória na GPU.

```
sumKernel<<<1, nElements/2>>>(thrust::raw_pointer_cast(&vector[0]),
                                        nElements, nSteps);
```

são passados o ponteiro do início do vetor, o seu tamanho, o número total de etapas, e entre `<<<numeroDeBlocoDeThread, numeroDeThreadPorBloco>>>`, nesse caso, 1 bloco de threads, com 8 threads. Dentro do kernel, para a seguinte instrução,

```
int threadId = (blockIdx.x * blockDim.x) + threadIdx.x;
```

tem-se `0 ≤ threadId < 8`, `threadId` é o índice da thread em execução. O laço `while` é executado simultaneamente pelas threads, e cada uma acessa duas posições do vetor e soma os conteúdos armazenando o resultado na posição mais à esquerda. Observe que, a declaração do kernel é precedida pela anotação `__global__` , isso é obrigatório, senão terá-se um erro de compilação.

```
__global__ void sumKernel(int *vector, int nElements, int nSteps)
```

Por convenção coloca-se um prefixo **d**_ no nome de variáveis de dispositivo, e **h**_ no nome de variáveis de host. Entenda por variável de dispositivo, uma variável alocada na memória da GPU, e variável de host, uma que é alocada na memória do computador; por exemplo, `thrust::device_vector<int>A(N);` aloca um vetor na GPU e `int soma;` aloca um inteiro na memória do PC.

4. DESENVOLVIMENTO

Neste capítulo, é, de fato, abordado o desenvolvimento (a parte prática) do estudo. Na seção 4.1, é dada uma *overview* sobre alguns algoritmos implementados em GPU; Na seção seguinte, a 4.2, são apresentados os algoritmos de ordenação e a abordagem usada nas implementações; a seção 4.3, faz uma exposição dos protótipos das funções implementadas; e por fim, a seção 4.4, trata dos testes dos algoritmos implementados.

4.1 Algoritmos implementados em GPU

Esta seção é baseada, na maior parte, no livro CUDA Programming de COOK, logo, apenas parágrafos retirados de outras bibliografias são referenciados.

A seleção de algoritmo para uma execução em GPU pode ser desafiadora, uma vez que envolve fatores que não, normalmente, são precisos em computação em CPU. Um bom algoritmo de CPU pode não, necessariamente, ser bom para a GPU; pois esta, por sua vez, contém suas próprias dificuldades. Logo, para ter uma melhor performance, é preciso conhecer o hardware (conhecimento do hardware é requerido para o processamento paralelo). Antes de escolher um algoritmo para um determinado uso em GPU, deve-se considerar os seguintes pontos:

- Decomposição do problema em blocos, e a particionamento dos blocos em threads;
- Como as threads acessarão a memória, e qual padrão de acesso usar;
- Identificar se há oportunidades de reuso dos dados, e como será realizado;
- Qual será o trabalho total do algoritmo na GPU, e qual será o ganho comparado à execução sequencial

4.1.1 Algoritmos de ordenação implementados em GPU

No dia a dia, algoritmos de ordenação são largamente usados em aplicações de software, com diversos propósitos. Entre estes, pode-se citar Bubble Sort, Rank Sort, Merge Sort, por exemplo; cada um com suas complexidades, que seja de implementação, de tempo de execução, de espaço necessário de memória, entre outros. Cada um leva um tempo diferente para processar um vetor de entrada com N elementos, e quanto maior for o tamanho do vetor, mais tempo precisará um algoritmo para ordená-lo. Por outro lado, se os mesmos pudessem ser acelerados usando CUDA, isto é, programá-los para serem executados em GPU, e se pudessem aproveitar um extenso paralelismo, ter-se-ia, em compensação, um ganho significativo em desempenho (VAIDYA, 2018).

Um algoritmo de ordenação deve, recebendo uma entrada V com N elementos aleatoriamente ordenados, produzir uma saída com os N elementos em uma determinada ordem, crescente ou decrescente. O enfoque principal da ordenação é garantir a minimização do número de leituras e escritas na memória. A maioria dos algoritmos, atualmente em uso, é multipassos; ou seja, em sua execução, o mesmo lê cada elemento de V, M vezes, o que, obviamente, não é bom. Entre os diversos algoritmos de ordenação existentes, alguns são fáceis de implementar em GPU e têm um desempenho eficiente; entretanto, alguns não respondem tão positivamente ao switch de CPU para GPU.

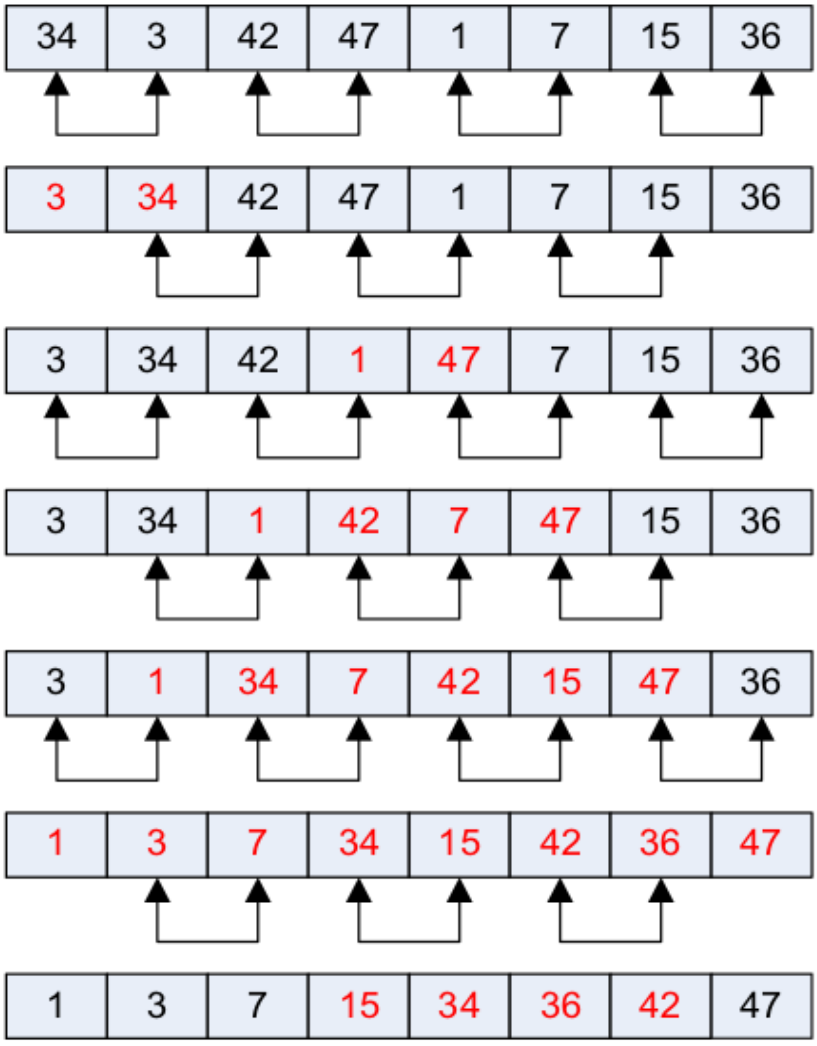
4.1.1.1 Odd/Even Sort (Ordenação Impar/Par)

O método de ordenação Ímpar/Par é uma variação de Bubble sort. O algoritmo Bubble sort processa a ordenação selecionando sempre, para um vetor A com N elementos, o primeiro elemento de A e indo comprando e trocando, se for necessário, com o próximo; isto é, comparar o elemento A_i com o elemento A_{i+1}, trocando os, caso necessário, e continuar até o elemento A_{n-1}, para $0 \leq i \leq N - 1$. O Odd/Even Sort, por sua vez, amplia essa metodologia de uma forma em que a ordenação seja

processada em paralelo usando P threads independentes. Sendo $\mathbf{P} = \frac{N}{2}$, ou seja, o número de threads é a metade do número de elementos presentes no conjunto.

O Odd/Even Sort seleciona, conforme mostrado da figura 11, cada índice par, e compara o seu valor com o elemento de índice ímpar maior e adjacente ao mesmo. Se o elemento de índice par for maior do que o de índice ímpar, então os mesmos são trocados; e esse processo continua iniciando com os índices ímpares, comparando-os com os elementos de índice pares maiores e adjacentes. Obviamente, a execução para quando não houve troca em uma iteração, ou seja, o conjunto já se encontra ordenado. Vale notar que, o algoritmo considera o menor índice sendo 0 e é par.

Figura 11: Processo de ordenação de Odd/Even Sort



Fonte: COOK, 2013

Essa abordagem pode parecer atraente, quando pensar em implementar $\frac{n}{2}$ threads em CPU, conceitualmente, é simples; mas há alguns pontos que devem ser considerados para uma implementação em GPU desse último. O primeiro problema é que o Odd/Even Sort é projetado para sistemas paralelos onde um processador individual de elementos possa trocar dados com o seu vizinho imediato. Neste caso, é apenas requerida uma conexão com seu vizinho esquerdo e direito. Para contornar esse problema, é usada memória compartilhada na abordagem com GPU. Segundo, é que pode haver conflitos no uso da memória compartilhada. Por exemplo, a thread 0 pode precisar ler a posição A_0 e A_1, em seguida escrever em A_0; a thread 1 pode precisar, por sua vez, ler a posição A_1 e A_2, em seguida escrever em A_1; assim por diante, ou seja, haverá escrita e leitura em um mesmo local simultaneamente, causando uma sequência de problema. Em um sistema de computação 1.x esse padrão de acesso é terrível, pois envolve várias buscas de 32 bytes. No entanto, em um sistema 2.x os acessos buscam no máximo duas linhas de caches, e dados obtidos no ciclo par, provavelmente, estarão disponíveis para o ciclo ímpar, e vice-versa. Além de ter uma quantidade significativa de reutilização de dados, esse sistema permite escolher entre cache e/ou memória compartilhada. Contudo, para o sistema 1.x a memória compartilhada seria, provavelmente, a única escolha devido à junção pobre.

4.1.1.2 Quicksort

O Quicksort, no processamento sequencial, pode ser o preferido por usar-se a estratégia de divisão e conquista; logo, pode ser uma boa escolha para uma abordagem paralela. Contudo, por ser recursivo, na essência, é possível a sua implementação dessa forma apenas em sistemas de computação CUDA 2.x, isto é, em uma GPGPU com suporte para a versão 2.x de CUDA. O paralelismo típico usa um nova thread para cada partição de dados, nesse caso não é necessário informar, de início, o número de threads a serem usados; enquanto, o modelo atual de CUDA requer a especificação do número total de threads na execução do kernel. A divergência de ramo causada pelo particionamento dos dados não é desejável no uso da GPU. É claro que existem meios de abordar esses problemas, no entanto, são o

que faz do Quicksort não ser uma boa escolha para um sistema GPU pre-Kepler GK110 e Tesla K20. O fato de que o melhor algoritmo sequencial não seja o melhor para uma abordagem paralela, é normal; cabe ao desenvolvedor escolher qual mais se encaixa na resolução do problema.

4.1.1.3 Merge sort

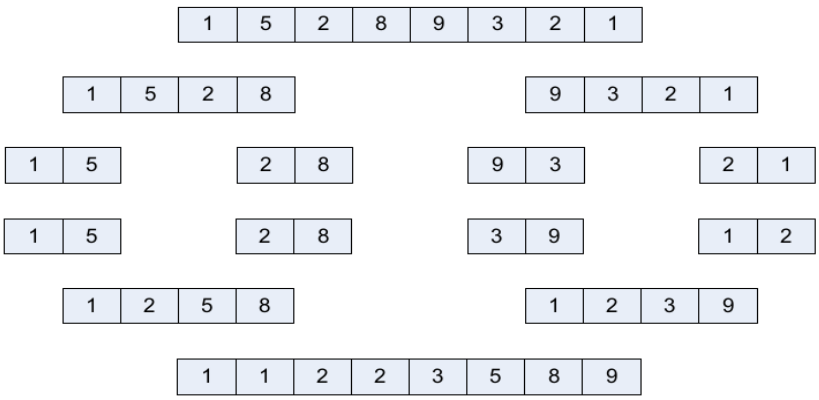
É comum o uso do Mergesort no processamento paralelo. Assim como o Quicksort, a sua natureza recursiva faz com que seja possível implementar o mesmo em uma arquitetura CUDA 2.x; sua forma de trabalhar é semelhante à da computação sequencial. De fato, o algoritmo particiona o conjunto em pequenos subconjuntos até chegar a partes de tamanho dois (2) para serem ordenadas. Toda parte ordenada é, então, mesclada para, enfim, produzir a saída ordenada.

Todo algoritmo recursivo, em algum momento da sua execução, tem um conjunto de *n* elementos. Na implementação em GPGPU, o tamanho de bloco de threads ou de o tamanho de um warp (empenamento) é o mais interessante para esse *n*; logo, para implementar um algoritmo recursivo em GPGPU, o que é necessário é quebrar os dados em blocos de 32 ou mais elementos no menor caso de *n*. Por exemplo, usando o mergesort, se tiver um conjunto A= {1,5,2,8,9,3,2,1}, obviamente, de tamanho 8 (oito), pode-se quebrá-lo em dois subconjuntos de dados de tamanho 4 (quatro), Assim: {1,5,2,8} e {9,3,2,1}, pode-se, então, usar 2 (dois) threads P_1 e P_2 para ordenar cada parte. Se, nesse caso, quiser quebrar os dois subconjuntos em quatro, dessa forma: {1,5}, {2,8}, {9,3} e {2,1}, isso gera um caso trivial com quatro threads, cada uma compara um par de elementos e troca, se necessário; no final tem-se {1,5}, {2,8}, {3,9} e {1,2}. Vale ressaltar que, o número máximo de threads necessário, nesse caso, é $\frac{n}{2}$.

Agora, a fase de mesclagem é um problema clássico, quando se usa o mergesort. Nesta fase, a saída é produzida (ou montada) movendo cada menor elemento de cada parte para a mesma repetidamente, até consumir (mover) todos os

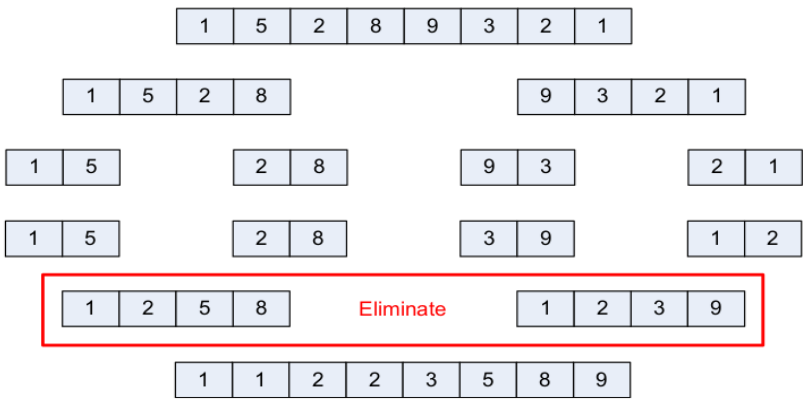
elementos da entrada. Em uma abordagem tradicional, teria {1,5}, {2,8}, {3,9} e {1,2} mesclados em {1,2,5,8} e {1,2,3,9}, e em seguida esses dois últimos mesclados em {1,1,2,2,3,5,8,9}, como mostrar a figura 12. Como, à medida que uma etapa de mesclagem é concluída, o número de paralelismo (*threads*) disponíveis cai pela metade; se *n* for pequeno, uma alternativa seria, simplesmente, pular qualquer etapa de mesclagem intermediária, e agir direto sobre os *n* conjuntos e colocar o valor no conjunto de saída corretamente, como mostrar a figura 13. O problema é que a etapa de descolagem marcada para ser pulada na figura 13, é feita, normalmente, por duas threads. Qualquer coisa abaixo de 32 threads significa que está usando menos de um warp, isso gera ineficiência no processamento da GPU.

Figura 12: Mergesort normalmente executado



Fonte: COOK, 2013

Figura 13: Mostra a etapa de mesclagem a ser pulada em uma abordagem com *n* pequeno.



Fonte: COOK, 2013

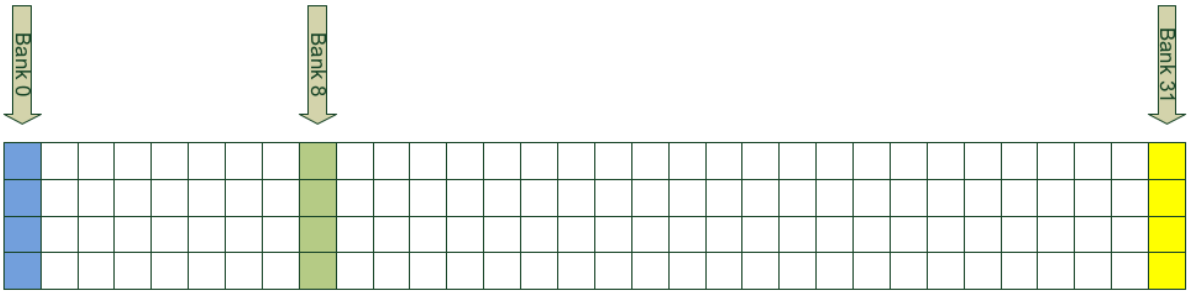
A maior desvantagem da abordagem anterior é que precisa ler o primeiro elemento de cada parte ordenada. Se tiver conjuntos de 64K, são 64K de leituras ou 256MB de dados que precisam ser buscadas na memória, o que, obviamente, não é uma boa solução quando o número de partes é muito grande. Uma abordagem para uma solução melhor, é limitar o número de recursão originalmente feito, parando no número de threads em um warp, 32 elementos conforme a figura 14, em vez de chegar até 2 elementos por subconjuntos, como é feito tradicionalmente. Isso reduzirá o número de partes, de 64K mencionado anteriormente, para apenas 4K; além de que o número máximo de paralelismo disponível passará de $\frac{n}{2}$ para $\frac{n}{32}$. Além disso, para uma GPU como a GTX580, em que cada SM da arquitetura Fermi pode executar até 48 warp, vários blocos precisarão ser iterados, o que, de fato, permite problemas de tamanhos menores, e consequentemente, ganho de speedup em hardwares mais avançados.

Figura 14: Subdivisão do conjunto até o tamanho de um warp, para poder usar 32 threads.

Fonte: COOK, 2013.

Uma atenção sobre a memória compartilhada. Tem-se 32 threads em um warp, no entanto, se mais de uma acessam uma mesma posição simultaneamente, pode haver conflito; e também, se alguma thread se perder no fluxo de execução, pode causar uma execução a $\frac{1}{32}$ da velocidade, no pior caso. Uma organização de conjunto de dados em linha de 32 elementos (observe a figura 15) na memória compartilhada e o acesso uma coluna por thread, pode conter (eliminar) os conflitos. A procura por mais amplas informações sobre gerenciamento de conflitos, é deixada ao leitor, caso tenha interesse.

Figura 15: Organização e acesso em coluna por thread da memória compartilhada.



Fonte: COOK, 2013.

4.2 Algoritmos implementados

São implementados 4 algoritmos de ordenação, usando em conjunto as APIs apresentadas anteriormente. Dois desses algoritmos são implementados com uma abordagem que permite enxergar o controle sobre a execução, e outros dois são construídos de uma maneira mais transparente, que será apresentada posteriormente. Para se ter noção da execução de algoritmos em GPU, é comprado o tempo de execução dos mesmos ao tempo de execução de uma implementação sequencial em C de cada um. Mas, vale ressaltar que, dependendo da abordagem, cada algoritmo implementado pode alcançar um speedup maior ou menor do que será apresentado neste estudo.

4.2.1 Insertion Sort

A ordenação por inserção arranja uma sequência inserindo um a um os elementos em suas devidas posições. Em uma implementação convencional tem-se a complexidade de tempo $O(n^2)$ no pior caso, e $O(n)$ no melhor caso, isso acontece quando a lista já se encontra ordenada (SZWARCFITER; MARKENZON, 2010).

Para a GPU, é usada uma abordagem paralela na qual é feita uma busca pela posição correta da chave com complexidade $O(1)$, o rearranjo dos k chaves

ordenadas, também com custo $O(1)$, assim como a inserção, isso faz com que o algoritmo tenha um custo $\theta(n)$. O algoritmo assume que tenha, para um conjunto N chaves, $N-1$ processadores, ou seja, no nosso caso $N-1$ threads; e também, o mesmo requer uma lista de tamanho $N+2$, sendo a primeira e a última posição da lista setadas com infinito negativo e infinito positivo, respectivamente. Na busca cada thread T_k compara os elementos A_k e A_{k+1} com a chave X a ser incluída, e a T_k cujas chaves $A_k \leq X < A_{K+1}$ escreve seu índice acrescentado de um (1) em uma variável. Em seguida, é processado o rearranjo dos elementos posteriores ao índice retornado, neste caso, cada T_k lê a posição A_k e escreve o conteúdo em A_{k+1}. Por fim, é feita a inserção no índice devolvido pela busca. Vale ressaltar que, o algoritmo baseia-se sobre uma sublista de tamanho 2 ordenada, sendo as duas primeiras posições do vetor de entrada.

4.2.2 Merge

O algoritmo de classificação por mesclagem trabalha com duas listas ordenadas $A(a_1 < a_2 < a_3 < \ldots. < a_n)$ e $B(b_1 < b_2 < b_3 < \ldots. < b_m)$ que serão mescladas em apenas uma lista C de tamanho $n+m$ ordenadamente. Nesta abordagem, para um a_i, em A, é chamado de $Rank(a_i, B)$ o número de elementos de B menores ou iguais à a_i, de maneira similar, para um b_i em B, o $Rank(b_i, A)$ é o número de chaves de A cujos valores são menores ou iguais a b_i. O processo é o seguinte:

1. Calcula-se o $Rank(a_i, B)$ e $Rank(b_i, A)$, para cada $a_i \in A$ e $b_i \in B$
2. Calcula-se o $Rank(a_i, A)$ e $Rank(b_i, B)$, isto é, o rank de cada elemento em relação ao próprio vetor.
3. Calcula-se a soma de $a_j + aa_j$ para cada $a_j \in Rank(a_i, B)$ e cada $aa_j \in Rank(a_i, A)$, aqui chama-se o resultante de AA; repete-se esta etapa para $Rank(b_i, A)$ e $Rank(b_i, B)$, o resultante é chamado de BB.
4. Processe a ordenação da seguinte forma, para cada $b_i \in BB$, $C[b_i] = B_i$ e para cada $a_i \in AA$, $C[a_i] = A_i$.

Por exemplo:

```
A[1, 4, 10, 30]                e          B[11, 15, 24, 35]
RA = Rank(a_i, B) = [0, 0, 0, 3]          RB = Rank(b_i, A) = [3, 3, 3, 4]
RAA = Rank(a_i, A) = [1, 2, 3, 4]         RBB = Rank(b_i, B) = [1, 2, 3, 4]
AA = RA_i + RAA_i = [1, 2, 3, 7]          BB = RB_i + RBB_i = [4, 5, 6, 8]

para a_i em AA e b_i em BB:
C[a_i] = A_i = [1, 4, 10, , , , 30, ]
C[b_i] = B_i = [1, 4, 10, 11, 15, 24, 30, 35]
```

e assim é preenchida a lista de saída. O algoritmo, nesta abordagem, assume que tenha-se, para duas listas de tamanho N e M, `max(N, M)` processadores (threads). A seguir, os protótipos dos procedimentos.

Está claro que, em uma versão sequencial deste algoritmo, será preciso um tempo de $O(n+m)$. Contudo, na implementação paralela feita, se chamar de k o número de elementos menores ou iguais a uma determinada chave em qualquer uma das listas, será necessário $O(k)$ tempo para determinar o rank dessa dita chave. Importante notar que, o cálculo dos ranks é feito em paralelo, ou seja, o rank de todos os elementos da lista é calculado simultaneamente; como é calculado o rank para 4 listas, então tem-se $O(k) + O(k) + O(k) + O(k)$. A somatória de `rankProper[i] = rankCompared[i] + rankProper[i]` para $0 \leq i < n$, gasta um tempo constante $O(1)$, pois é feito em paralelo sem repetição, cada thread T_i acessa uma posição de cada lista, soma os valores e grava no índice i da lista de saída o resultado. A montagem da sequência ordenada gasta, também, tempo constante de $O(1)$. Portanto, Tem-se, no máximo, o algoritmo de classificação por mesclagem, nesta implementação, gasta tempo $O(k)$. Contudo, como as listas estão inicialmente ordenadas, pode-se usar a busca binária para determinar o `Rank (x_i, V)`, que daria um ganho de desempenho, pois a busca gastaria $O(\log n)$, e consequentemente, o algoritmo precisaria no máximo tempo $O(\log n)$.

4.2.3 Paralelismo dinâmico

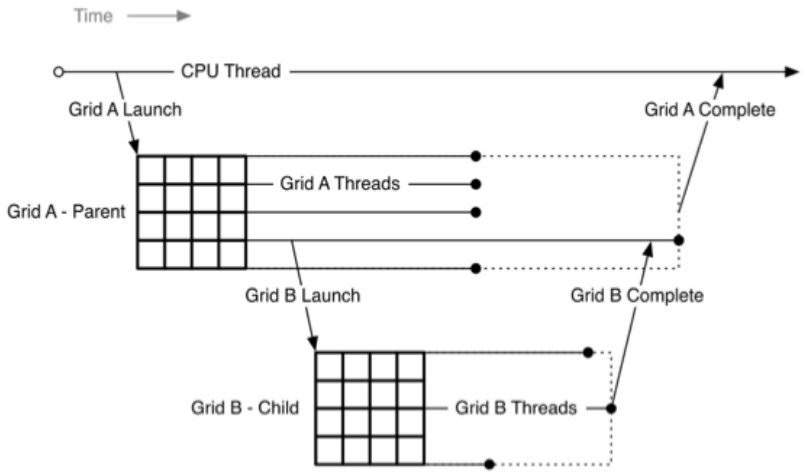
Antes de apresentar as duas implementações restantes, precisamos introduzir o conceito de paralelismo dinâmico de CUDA.

Esta subseção baseia-se nas referências de um artigo publicado em 2014 no NVIDIA DEVELOPER.

4.2.3.1 Aninhamento de grade e sincronização

No modelo de programação CUDA, uma grade é um grupo de blocos de threads que executa um kernel. Utilizando o paralelismo dinâmico, a grade mãe lança kernels chamados grade filha que herda certos atributos e configuração da mãe. Em uma execução, as grades filhas sempre terminam antes das grades mãe, mesmo se não houver, explicitamente, um ponto de sincronização. Em um caso onde a grade mãe precisa de um resultado calculado por uma filha, para fazer algum trabalho, a mãe deve assegurar-se que toda grade filha termine a sua execução antes de prosseguir, para isso, são especificados pontos de barreira chamando a função `cudaDeviceSynchronize()`. A figura 16 mostra o acompanhamento de uma grade mãe à execução de uma grade filha. Note que, na grade filha, todas as threads terminam a sua execução antes da grade mãe retomar o controle da execução. Uma observação importante, é que a grade mãe configura a filha explicitamente com o número blocos e threads que a mesma deve usar, caso não esteja configurada, essa última executará todos os blocos disponíveis na grade mãe. É importante ressaltar que, em uma grade mãe, alguns tipos de ponteiros não podem ser passados à uma grade filha, como ponteiro de variáveis compartilhadas declaradas com `__shared__`; memórias locais; assim como ponteiro de pilha.

Figura 16: Mostrando o mecanismo de lançamento de grade de threads filhas por um kernel pai.



Fonte: NVIDA DEVELOPER, 2014

4.2.3.2 Streams e eventos do dispositivo

Dentro de um bloco de threads, grades lançadas são executadas sequencialmente por padrão. Quando se desejar que essas grades executam concorrentemente, deve-se, explicitamente, configurar o kernel para a criação de novos fluxos. Para essa operação, a única forma existente em CUDA, atualmente, é através da criação de stream que é feita conforme o código abaixo.

```
cudaStream_t stream;
cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
```

Esse código cria um novo fluxo de execução, para uma chamada recursiva de um kernel isso é muito útil. O segundo parâmetro especifica que a criação do fluxo não deve oferecer suporte à sincronização, mesmo sem ponto explícito. No entanto, deve-se atentar a observação, fluxos criados em blocos de threads diferentes são considerados diferentes, logo, a execução simultânea não é uma garantia. Ela é garantida apenas quando são utilizados os recursos da GPU de forma a aproveitar o maior nível de paralelismo.

Nas chamadas recursivas de um kernel, usa-se um fluxo diferente para cada invocação deste mesmo. Por exemplo, na implementação dos algoritmos Quicksort e Mergesort feita neste estudo. Assim, são criados dois fluxos, sendo um para execução da chamada à esquerda e um outro para a invocação à direita, como segue:

```
 cudaStream_t leftStream, rightStream;
//chamada à esquerda
cudaStreamCreateWithFlags(&leftStream, cudaStreamDefault);
 quickSortKernel<<<1, 1, 0, leftStream>>>(vector, left1,   right1);
 cudaStreamDestroy(leftStream);

//Chamada à direita
 cudaStreamCreateWithFlags(&rightStream, cudaStreamDefault);
  quickSortKernel<<<1, 1, 0, rightStream>>>(vector, left2, right2);
 cudaStreamDestroy(rightStream);
```

Neste caso, os streams são criados usando o modo padrão de criação `cudaStreamDefault` ao invés do modo não bloqueante. Um fluxo criado, não deve ser utilizado depois da execução do bloco de threads, por isso é chamado a função de destruição de stream `cudaStreamDestroy` para limpar as configurações do mesmo. Não há uma forma de sincronizar um fluxo atualmente, e também não é necessário, uma vez que uma chamada de `cudaStreamDestroy` garante que os recursos sejam liberados automaticamente, ao concluir a tarefa. Contudo, caso esteja desejada uma sincronização explícita, a forma de fazer é garantir que todas as threads terminam seu trabalho, chamando `cudaDeviceSynchronize()`.

4.2.3.3 Profundidade de recursão e limite do dispositivo

O paralelismo dinâmico é, largamente, usado em algoritmos recursivos, a esses, no geral, é associada uma profundidade de recursão. A abordagem de paralelismo dinâmico é interessante pois nas grades de threads filhas, como mencionado anteriormente, pode-se ter uma execução com n threads sem ter especificado de maneira explícita. Contudo, é mais difícil de gerenciar, pois não se tem controle sobre as threads em execução. Dois mecanismos de profundidade que são amplamente usados:

- Profundidade de aninhamento, que é o aninhamento mais profundo de lançamento de grade recursiva.
- Profundidade de sincronização, que é o nível mais profundo de recolhimento no qual `cudaDeviceSynchronize()` é chamada. Esse limite de profundidade é indicado ao chamar-se a função `cudaDeviceSetLimit(cudaLimitDevRuntimeSyncDepth, limit).` sendo `cudaLimitDevRuntimeSyncDepth` o `limit` atual da GPU e `limit` o novo limite associado.

No geral, a profundidade de sincronização é um a menos que a profundidade de lançamento recursivo.

O limite de hardware na profundidade de aninhamento é, na atualidade, no máximo 24; isto é, se em uma função recursiva, são feitas mais de 24 chamadas sem uma invocação à `cudaDeviceSynchronize()`, terá-se um erro, pois será estourado o limite máximo. O número de grades filhas pendentes, que define número de grades em execução, suspensas e em espera, é outro limite importante, atualmente, é no máximo 2048, logo, se ultrapassar esse extremo, tem-se um erro.

Agora que entendemos, de maneira simplificada, o que é paralelismo dinâmico com CUDA, podemos prosseguir apresentando os dois últimos algoritmos.

4.2.4 Quicksort e Mergesort

Quicksort e Mergesort, em uma implementação sequencial conseguem ordenar uma lista com uma complexidade $O(n\ log\ n)$. No entanto, o Quicksort pode alcançar um desempenho igual aos piores algoritmos de ordenação $O\ (n^2)$, dependendo da ordem dos elementos da lista, inicialmente, e a escolha do pivô (SZWARCFITER; MARKENZON, 2010).

O Quicksort em uma implementação recursiva, como já explicado anteriormente, não é eficiente em GPU, se a abordagem leva em conta o controle

sobre as threads em execução, isto é, controlar conjuntos de threads em execução. Essa desvantagem está relacionada à escolha do pivô, que não segue uma divisão uniforme em relação ao tamanho dos subvetores, ou seja, em alguns casos tem-se uns subvetores de tamanho X e outros com tamanho Y, sendo X ≠ Y. Também, já foi explicado o funcionamento do Mergesort. Na implementação desses últimos é usada a abordagem sugerida na explicação do Mergesort, na seção anterior, ou seja, a profundidade máxima de lançamento de recurso utilizada, é o tamanho de um warp (32 threads). O esquema usado é o seguinte:

- Nos dois algoritmos, a lista é dividida até as sublistas terem tamanho de 32 elementos;
- A cada divisão, são criados dois fluxos de execução (esquerda e direita), isso faz com que seja lançado um novo bloco de threads a cada particionamento do conjunto.
- Quando atingir, uma sublista, o tamanho de um warp, é chamado um algoritmo que ordene a mesma; o algoritmo utilizado é o odd/even sort que já foi explicado anteriormente.
- E, voltando na recursão, o Mergesort chama o kernel `intercalatekernel` que faz a mesclagem das sublistas ordenadas. O Quicksort, por sua vez, a partir do último lançamento de recursão, saí com a lista ordenada.

Essa abordagem deixa claro que a divisão da lista não gaste $O(log\ n)$, que é a altura de uma árvore binária completa, pois a distribuição não chega até sublistas de tamanho 1. Além disso, a ordenação de cada sublista, no máximo gasta $O(k)$, sendo k o tamanho de um warp, isto é, 32 chaves por subvetor. Segundo Szwarcfiter e Markenzon, o número de nós folha de uma árvore binária é $\frac{n+1}{2}$, pois a mesma tem $n + 1$ subárvores vazias. Logo, podemos dizer que o algoritmo que ordena as partes da lista é chamado q vezes, onde $q < \frac{n+1}{2}$, a recursão gasta $O(log\ n - log\ 32)$, no caso do mergesort a intercalação gasta no máximo $O(n)$, portanto, a execução do Quicksort e Mergesort, nesta abordagem, tem uma complexidade diferente de $O(n\ log\ n)$. Não foi o foco aqui, apresentar uma análise completa da complexidade das implementações, mas sim o tempo de execução de cada.

4.3 Protótipos das funções/procedimentos implementados

No arquivo `global.h`, são definidos protótipos das funções implementadas neste estudo, seja elas utilitárias, kernels ou funções host. A seguir, são apresentadas cada uma delas além das constantes `MAX_RECURSION` que define a profundidade máxima do dispositivo (GPU), e `WARP_SIZE` que define o tamanho de warp (32 threads).

```
 #define MAX_RECURSION 24
 #define WARP_SIZE 32

//kernels utilitários usadas por outros kernels
 __global__ void vectorPrinter(int *, int, int);
 __device__ void swapKernel(int *, int *);
 __device__ void oddEvenSortKernel(int *, long, long);

//Implementação do Quicksort
 __device__ void pivotMedianOf3Kernel(int [], long, long, long *);
 __global__ void quickSortKernel(int *, long, long);

//Implementação do Mergesort
 __global__ void intercalateKernel(int *, int *, long, long, long);
 __global__ void mergeSortKernel(int *, int *, long, long);

//Implementação do Insertion sort
 __global__ void indexSearchKernel(int *, int, int, int *);
 __global__ void pushElementsKernel(int *, int, int);
 __global__ void insertionSortKernel(int *, long, int *);

//Implementação da ordenação por mesclagem
 __global__ void rankCalculatorKernel(int *, int *, int *, int, int);
 __global__ void rankSumCalculatorKernel(int *, int *, int);
 __global__ void mergeSortKernel(int *, int *, int *,  int *, int *,
int, int);

 void mergeSort(int *, int *, int *, int, int);

 void thrustSort(thrust::device_vector<int>, float *);
```

- `vectorPrinter`, é um kernel que serve para imprimir as chaves de uma lista ou parte de uma lista, recebendo o ponteiro da lista, o índice inicial e o número de elementos.
- `swapKernel` permite a troca de dois elementos de um conjunto, recebendo como entrada o ponteiro de cada elemento. Note que, é um kernel tipo `__device__`, ou seja, este só pode ser chamado por um outro kernel.
- `oddEvenSortKernel`, que ordena uma lista, recebendo como parâmetros a lista, seu primeiro índice e o número de chaves presente na mesma.
- `pivotMedianOf3kernel`, processa a escolha de um pivô usando mediana de três, tendo a lista, o índice inicial, o índice final, e ponteiro da variável de retorno. Ele é também de tipo `__device__`.
- `quickSortkernel`, que implementa o algoritmo de ordenação rápida. Este kernel faz uso do `pivotMedianOf3kernel` para a escolha do pivô e do `oddEvenSortKernel` para ordenar as partes de tamanho de um warp como explicado anteriormente. São requeridos o ponteiro da lista, o seu índice inicial e o último para seu funcionamento.
- `intercalateKernel`, que intercala duas partes ordenadas de uma lista, o mesmo é chamado pelo mergesort recursivo. As entradas são a lista, uma lista temporária usada na intercalação, índice inicial da primeira parte, o índice do meio, e o último índice da segunda parte.
- `mergeSortKernel`, é a implementação do algoritmo recursivo da ordenação por intercalação. Para auxiliar no processamento da ordenação, são chamados os kernels `oddEvenSortKernel` para ordenar as partes de tamanho de um warp e o `intercalateKernel` para mesclar os subconjuntos ordenados. Este kernel recebe o ponteiro da lista a ser ordenada, o ponteiro de uma lista auxiliar, o seu índice inicial e o último.
- `indexSearchKernel`, é usado para fazer uma busca da posição de inserção de uma chave em uma lista ordenada. Este recebe os seguintes parâmetros: a lista, o seu tamanho, o número a ser inserido, endereço de uma variável de retorno.
- `pushElementsKernel` permite de reorganizar uma lista para uma nova inserção, puxando uma posição cada elemento a partir do índice da inserção

até o tamanho da lista. A lista, o seu tamanho e a posição da inserção são as entradas requeridas.

- `insertionSortKernel` implementa o algoritmo de ordenação por inserção na GPU. Como auxiliares, são chamados os kernels `indexSearchKernel` e `pushElementsKernel`. Este último recebe a lista e o número de chaves da mesma.
- `rankCalculatorKernel` faz o cálculo dos rankings dos elementos de um conjunto em relação um outro conjunto, tendo os dois conjuntos, o arranjo de retorno dos rankings, e o tamanho de cada conjunto de entrada.
- `rankSumCalculatorKernel` processa a somatória dos rankings das chaves de um arranjo em relação um outro, e os rankings de cada elemento do mesmo em relação a si mesmo. Os parâmetros são os seguintes: os dois conjuntos dos rankings, e o tamanho da lista em questão, que é também o tamanho dos arranjos de ranking.
- São implementadas duas abordagens de mesclagem de lista. A primeira é a implementação recursiva da ordenação por intercalação, e o kernel `mergeSortKernel` que intercala duas listas ordenadas, como explicado anteriormente. Este último recebe sete (7) parâmetros, sendo ponteiros de duas listas ordenadas A e B, ponteiro da lista de saída, dois arranjos de Ranking dos conjuntos A e B, o número de elementos de A e B.
- `mergeSort`, é uma função host, ou seja, ela é executada pelo processador host (CPU). Sua atribuição é chamar o kernel que processa o merging das listas ordenadas, `mergeSortKernel`, assim como os kernels que auxiliam o mesmo nesse processo, ou seja, o `rankCalculatorKernel` e o `rankSumCalculatorKernel`.

4.4 Testes e Resultados

Esta seção é composta por uma apresentação das configurações da GPU nos quais serão feitos os testes; captura de tela mostrando o funcionamento de cada algoritmo; tabelas e gráficos mostrando as médias do tempo de execução de cada

algoritmo em relação à implementação sequencial do mesmo; é apresentada a função `sort(...)` da API Thrust; e uma breve análise dos resultados.

4.4.1 Apresentação da GPU

Trabalharemos aqui com a placa ZOTAC GeForce GT 1030 arquitetura pela NVIDIA. A GeForce GT 1030 lançada em 2017, é uma GPU potente e rápida, capaz de acelerar o processamento gráfico; e portanto, melhorar a experiência no uso dos computadores. Sendo a primeira da arquitetura NVIDIA Pascal, essa última conta com recursos gráficos e tecnologias que fazem com que seja poderosa e permita um speedup considerável de desempenho. Esse modelo pode ser usado para acelerar o processamento de jogos, dispondo de uma variedade de drivers NVIDIA dos mais usados do mercado (NVIDIA, 2021). Uma imagem ilustrativa do produto, é mostrada na figura 17.

4.4.1.1 Especificações técnicas

A GeForce GT 1030 é alimentada por barramento PCI Express, facilitando a portabilidade do dispositivo para várias plataformas (ou sistemas operacionais) diferentes, sejam elas de 32 ou 64 bits. A poderosa placa de vídeo conta com chips com um total de 380 CUDA cores, com velocidade de clock base de 1227 MHz, mas que pode atingir 1468 MHz; memória GDDR5 de 2 Gigabytes com barramento de 64 bits, e velocidade de clock de 6 Gigabytes; duas saídas gráficas, sendo uma VGA e uma HDMI; suporte HDCP; consumo de energia baixa, sendo 30 watt; Além de suporte DirectX 12, OpenCL e OpenGL 4.5 (ZOTAC, 2021). É deixado ao leitor a procura de mais amplas informações, consulte o ZOTAC GeForce GT 1030 , caso seja necessário (ZOTAC, 2021).

Figura 17: A placa de vídeo ZOTAC GeForce GT 1030

Fonte: ZOTAC, 2021

4.4.2 Testes de funcionamento

Nesta seção são apresentadas algumas capturas de telas mostrando o funcionamento de cada algoritmo para um conjunto de 128 elementos. Dois pontos importantes:

- O número de processadores (threads) usado é sempre uma potência de dois, isto é, $P = 2^k$ processadores.
- O número de chaves da lista deve ser um múltiplo de P, ou seja, $N = X \cdot 2^k$, com $X \in \mathbb{N}$. Se o conjunto de chaves tiver o tamanho menor do que 2^k, deve-se preencher com mais infinito o final do mesmo até alcançar $X \cdot 2^k$ elementos.
- As duas primeiras condições são necessárias apenas para as implementações que não usam paralelismo dinâmico, mas são importantes.

Nas figuras consta a impressão da entrada no terminal antes e depois da ordenação. Observe as figuras 18, 19, 20 e 21.

Figura 18: Ordenação de 128 elementos na GPU com insertion sort

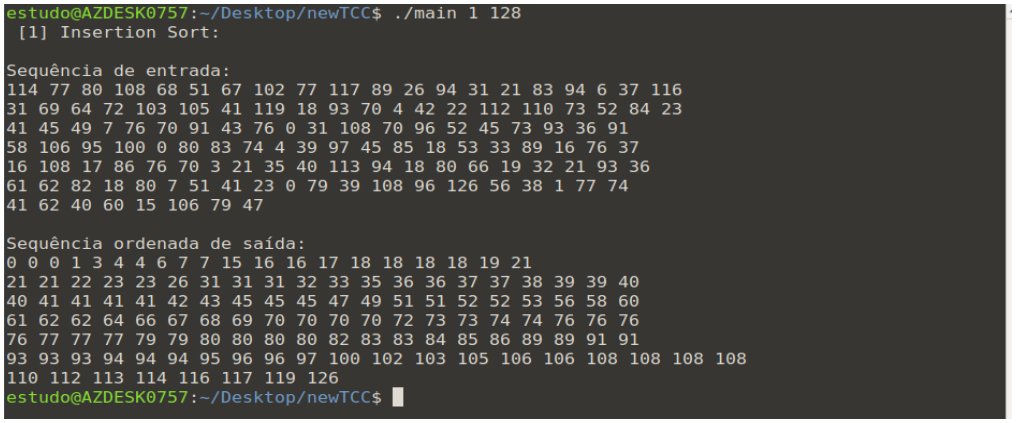


Figura 19: Ordenação de 128 elementos na GPU com Merge

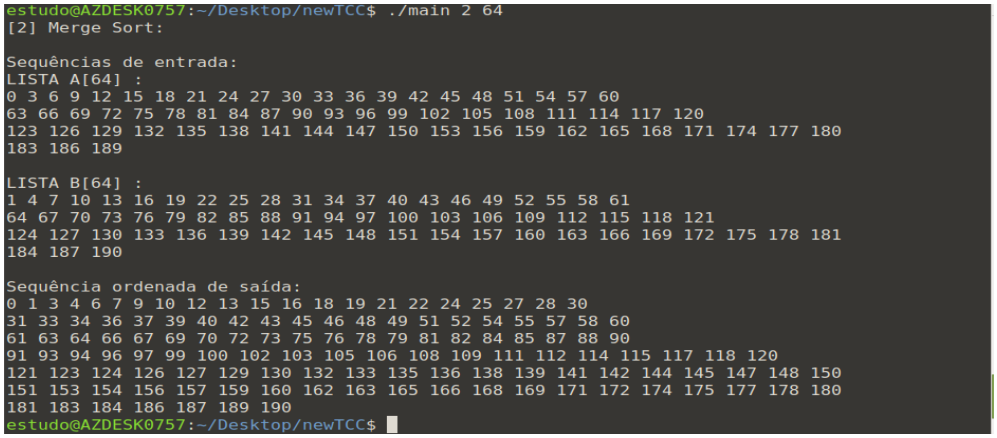


Figura 20: Ordenação de 128 elementos na GPU com Quicksort

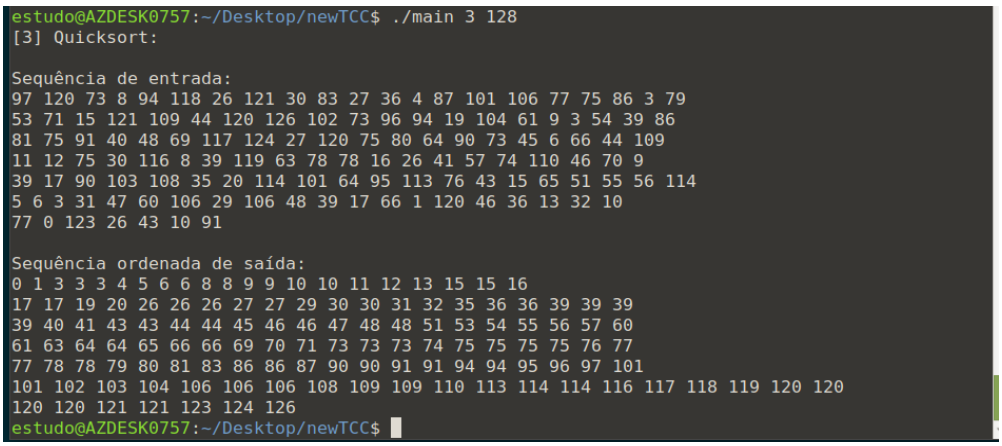
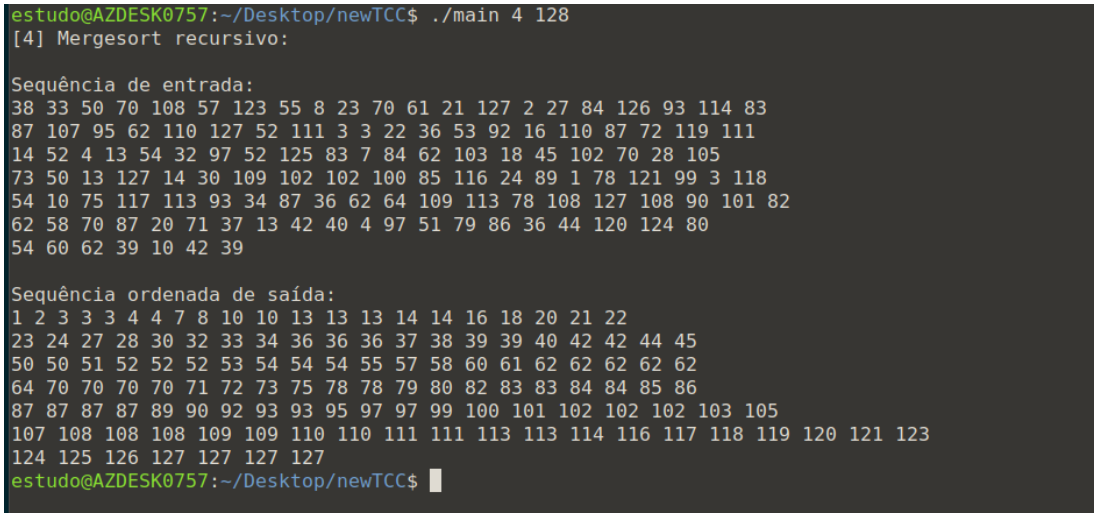


Figura 21: Ordenação de 128 elementos na GPU com Mergesort recursivo

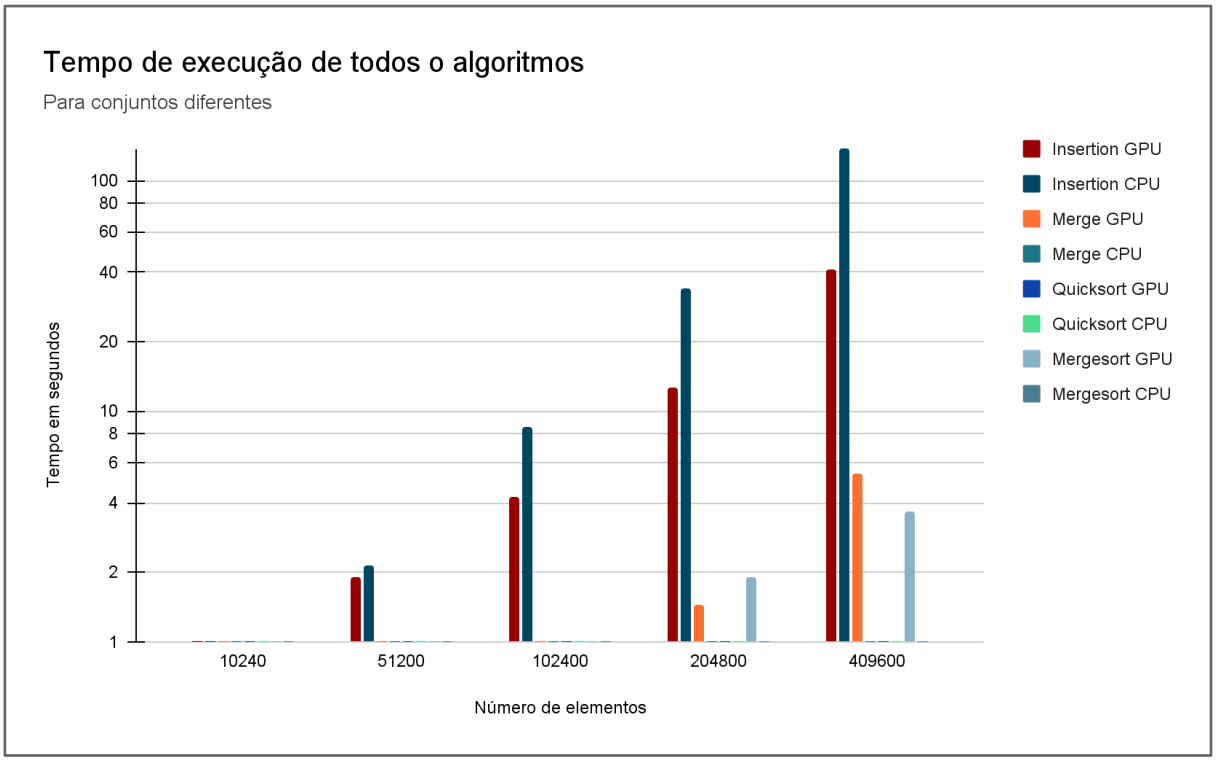


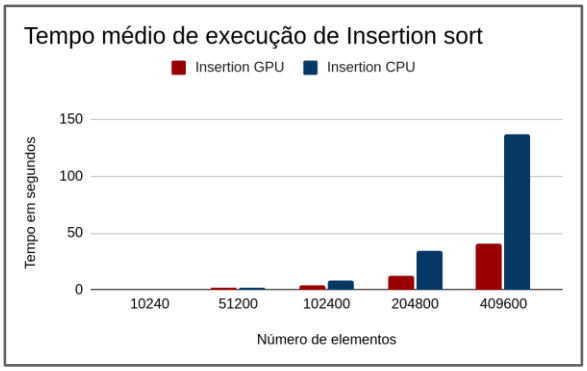
4.4.3 Tabelas e gráficos de tempo de execução

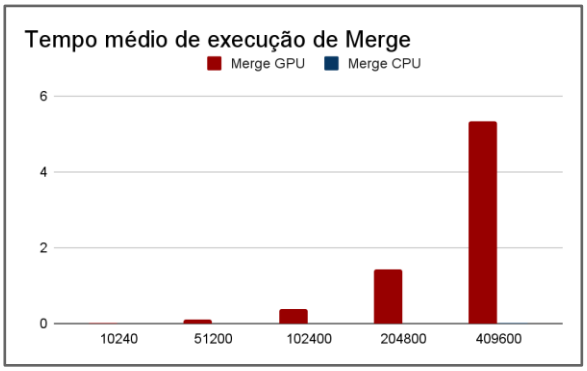
A tabela (1) e (2) mostram a média dos tempos de execução sequencial e paralela, respectivamente, de cada algoritmo para alguns conjuntos de chaves; e (3) e (4) mostram o tempo médio para conjuntos maiores, usando apenas os algoritmos implementados com paralelismo dinâmico, ou seja, Mergesort e Quicksort. Vale ressaltar que as chaves das entradas se encontram em ordem aleatória no início.

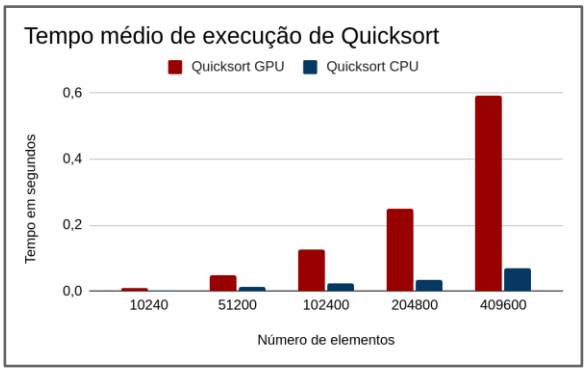
Algoritmos	**(1) Processamento em CPU** Entradas /Tempo em segundos				
	10.240	51.200	102.400	204.800	409600
Insertion Sort	0,088105	2,144405	8,569763	34,269021	137,304437
Mergesort	0,000115	0,000592	0,001182	0,002400	0,004788
Quicksort	0,002604	0,012491	0,022813	0,034519	0,070351
Mergesort recursivo	0,003353	0,013870	0,023831	0,045155	0,094178

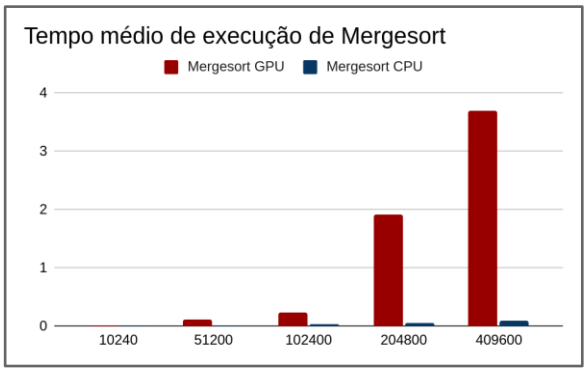
Algoritmos	(2) Processamento em GPU Entradas /Tempo em segundos				
	10.240	51.200	102.400	204.800	409.600
Insertion Sort	0.3953	1.9039	4.2359	12.5850	41.2291
Mergesort	0.0081	0.1106	0.4027	1.4487	5.3587
Quicksort	0.0076	0.0490	0.1263	0.2493	0.5906
Mergesort recursivo	0.0170	0.1060	0.2384	1.9182	3.6963





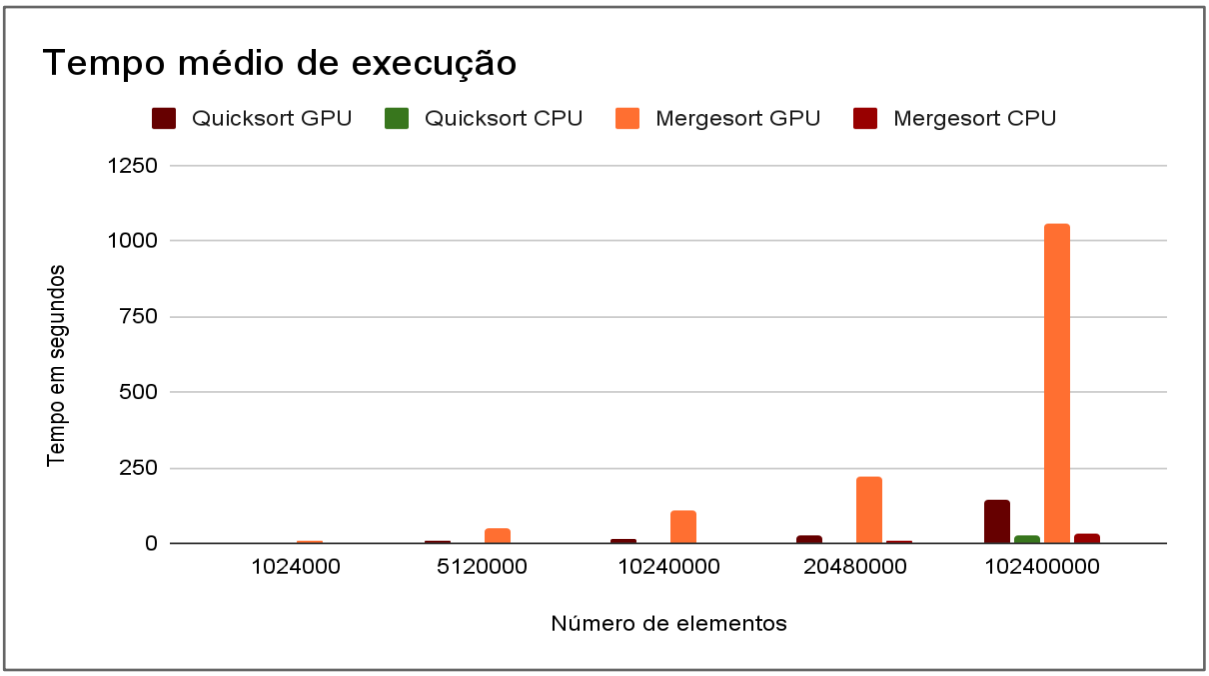


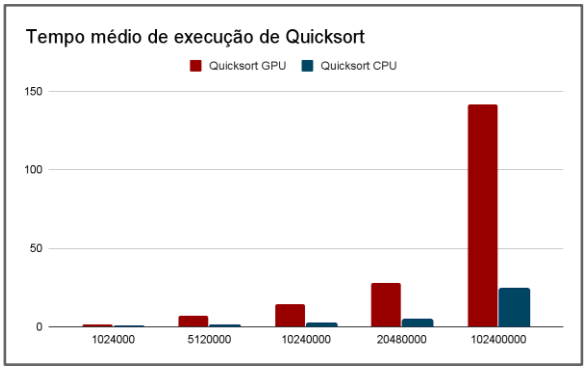


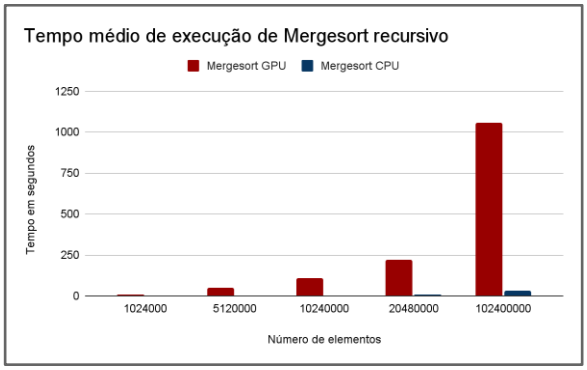


Algoritmos	**(3) Processamento em CPU** Entradas /Tempo em segundos				
	1.024.000	5.120.000	10.240.000	102.400.000	512.000.000
Quicksort	9,551344	52,232077	107,725273	222,240071	1057,735826
Mergesort recursivo	9,551344	52,232077	107,725273	222,240071	1057,735826

Algoritmos	(4) Processamento em GPU Entradas /Tempo em segundos				
	1.024.000	5.120.000	10.240.000	102.400.000	512.000.000
Quicksort	1,2330	7,2498	14,5356	27,8649	141,5221
Mergesort recursivo	9,5513	52,2320	107,7252	222,2400	1057,7358







4.4.4 CUDA Thrust Sort

Esta subseção é baseada na documentação da API Thrust.

Antes de prosseguir com a análise dos resultados, será feito uma apresentação de uma das funções de ordenação da API Thrust, `thrust::sort(...)`. A Thrust, como apresentada anteriormente, possui uma grande gama de funções de alto nível disponíveis para a maioria das operações. A `thrust::sort(...)` é uma função muito eficiente, que consegue ordenar, usando a GTX 480, 846 Milhões de chaves em ordem aleatória. Esta função pode ser chamada pela CPU, e também pela GPU, isto é, pode ser chamada em uma função host ou um kernel. A seguir o protótipo:

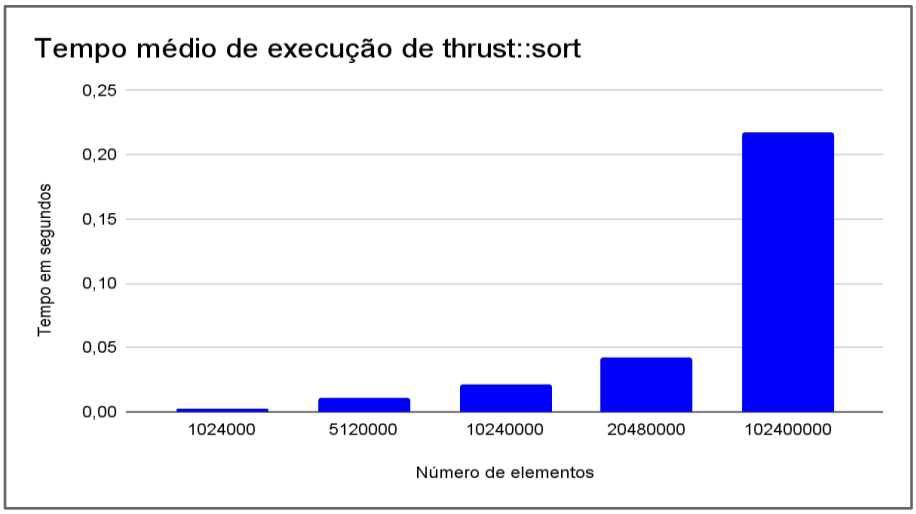
```
__host__ __device__ void thrust::sort(const
thrust::detail::execution_policy_base< DerivedPolicy > &exec,
      RandomAccessIterator first,
      RandomAccessIterator last,
      StrictWeakOrdering comp);
```

O primeiro parâmetro é a base da execução, host ou device; o segundo é o primeiro iterator da lista, terceiro é o último iterator, e o quarto é a base da comparação que, no exemplo abaixo, é ordem decrescente comparando inteiros. Note que, o primeiro e o último parâmetro não são obrigatórios. Observe o código abaixo:

```
//cria um vetor de 1024 elementos na memória do PC
 thrust::host_vector<int> host_vector(1024);
//preenche o vetor com valores randômicos
 thrust::generate(host_vector.begin(), host_vector.end(), rand);
//faz um cópia do vetor para GPU
 thrust::device_vector<int> device_vector = host_vector;
//ordena decrescentemente o vetor na memória do PC
 thrust::sort(thrust::host, device_vector.begin(), device_vector.end(),
thrust::greater<int>());
//ordena na ordem crescente na memória do PC
thrust::sort(thrust::host, device_vector.begin(), device_vector.end());
thrust::sort(device_vector.begin(), device_vector.end());
//ordena decrescentemente o vetor na memória da GPU
```

```
thrust::sort(thrust::device,device_vector.begin(),  device_vector.end(),
thrust::greater<int>());
```

A thrust possui algumas variações da `thrust::sort(...)`, como `thrust::key_sort(...)` que processa a ordenação de dois conjuntos usando a chave do primeiro especificado, entre outras. A `thrust::sort(...)` implementa Radix Sort para ordenar chaves de tipos primitivos, e Merge Sort para outros tipos de estrutura de dados. A Thrust, nessas implementações, faz uso da biblioteca `cub` que é uma coleção primitivos paralelos de todo bloco, primitivos paralelos de dispositivo, etc. A `cub` oferece componentes de software reutilizáveis de última geração para cada camada do modelo de programação CUDA. Para os mesmos conjuntos de dados utilizados nos testes das implementações do Quicksort e Mergesort, `N > 1M`, é apresentado um gráfico mostrando os tempos de execução da `thrust::sort(...)`.



4.4.5 Análise

Baseado nos gráficos e tabelas dos tempos mostrados, pode-se concluir que, a performance de um algoritmo na GPU, depende da abordagem usada na implementação do mesmo. Os dois pontos de vista codificados na ideia de mesclagem de lista, estão se comportando quase igualmente, com conjunto de dados

pequenos. Mas conforme aumenta o tamanho do conjunto, o Mergesort e o Quicksort se mostram mais eficientes. O Insertion Sort, obviamente, consegue um desempenho melhor em relação a sua versão sequencial, pois a sua implementação gasta $O(n)$; mas, não é a melhor escolha, quando se trata de arranjar conjuntos de grande tamanho. O Quicksort, mesmo não apresentando uma performance superior do que a sua melhor implementação sequencial, tem um comportamento, consideravelmente, satisfatório. Com o algoritmo dois (2), pode-se alcançar um melhor desempenho, se no cálculo dos ranks usa-se a busca binária, já que as listas estão, inicialmente, ordenadas. Portanto, o desempenho de um algoritmo não depende apenas do dispositivo (CPU ou GPU), mas, também, da implementação.

5. CONCLUSÃO

Na busca por melhor performance em processamento computacional nas indústrias, pesquisas universitárias e outras, as GPU estão sendo usadas com mais frequência a cada dia; pois esse dispositivo se mostra eficiente em várias abordagens, como em simulação na física e química, entre outras possibilidades. Uma das vantagens deste último é prover alto desempenho paralelo, isto é, permitir que os usuários tenham um processamento massivamente paralelo, com centenas de milhares de threads executando, simultaneamente, uma determinada tarefa. Logo, alcançar o melhor desempenho no uso de uma GPU é condicional à estrutura da tarefa; com estrutura da tarefa, queremos dizer as configurações da mesma. Portanto, deve-se responder às seguintes perguntas ao pensar em usar uma GPU em um trabalho qualquer: A tarefa é paralelizável? Há dependência de dados entre as partes da mesma? A paralelização deste último oferecerá um melhor tempo de execução do que o melhor algoritmo sequencial para o mesmo problema? entre outras perguntas, como relacionadas a complexidade de espaço, etc.

Conforme as empresas adotam a cultura de usar as GPUs para melhorar o poder de computação dos seus serviços, será preciso de profissionais capazes de programar esses dispositivos poderosos. Levando em conta a configuração dos cursos de computação, que sejam Sistemas de Informações, Ciência ou Engenharia da Computação, foi julgado importante um estudo que possa introduzir a programação desses últimos. Sendo, a ordenação de conjuntos, uma das tarefas diária na área da tecnologia; essa introdução à programação das GPUs foca na implementação de alguns algoritmos desses, com intuito de mostrar os passos para construir um processamento em GPU.

5.1. Limitação e vantagem

A implementação da ordenação por inserção e da ordenação por mesclagem de duas listas ordenadas, possui a limitação quanto ao número de threads disponível

no dispositivo, ou seja, o número de threads concorrentes que este pode lançar em uma tarefa. Por isso, conforme o aparelho que foi usado nesse estudo, essas implementações não podem ser usadas para ordenar listas de tamanho maior que 512000 (Quinhentos e doze mil) de chaves; pois a abordagem usada assume que tenha uma thread para cada elemento do conjunto, o que, geralmente, não ocorre. A principal vantagem desse comportamento implementado é que os dois são auto escaláveis, isto é, em um dispositivo com mais multiprocessadores, e consequentemente mais blocos de threads disponíveis, pode-se tranquilamente aumentar o tamanho da entrada sem necessidade de adaptar o código.

O Quicksort e o Mergesort, em contrapartida, conseguem ordenar conjuntos de tamanho maior. As principais desvantagens desses dois, são a limitação da profundidade do dispositivo, que é atualmente 24; a natureza recursiva; e a dependência de dados de uma etapa à outra, isto é, a posição correta de um elemento determinada no passo `k` depende da sua posição no passo `k-1`. Como, a maioria dessas operações é feita sequencialmente, então não há como alcançar um alto nível de paralelismo, o que faz com que os mesmos não sejam tão eficientes em GPU quanto são em CPU, como já foi explicado anteriormente. A vantagem mais óbvia desses, é o paralelismo dinâmico, ou seja, não são restritos em relação ao número de threads concorrentes que a GPU pode lançar.

5.2. Trabalhos futuros

Futuramente, algumas curiosidades que farão objeto da continuação desse estudo são listadas a seguir;

- Executar os mesmos testes em um dispositivo com perfil melhor do que o que foi usado. A RTX 3090, por exemplo.
- Testar conforme o item 1, com entradas de tamanho maior.
- Proceder ao tratamento de erros em tempo de execução, alocação, entre outros.
- Utilizar uma abordagem mais baixo nível nas alocações e gerações das entradas. E, implementar os mesmo sem recursos usando a API `cub`.

REFERÊNCIAS BIBLIOGRÁFICAS:

WEISKOPF, D. - GPU-Based Interactive Visualization Techniques, 2007

PATTERSON, D. A.; HENNESSY J. L. - Organização e Projeto de Computadores, Quinta edição, 2014

PATTERSON, D. A.; HENNESSY J. L - Arquitetura de Computadores: Uma abordagem quantitativa, Quinta edição, 2014

McCLANAHAN, C. - History and Evolution of GPU Architecture, 2010, url: http://www.mathcs.emory.edu/~cheung/Courses/355/Syllabus/94-CUDA/Docs/gpu-hist-paper.pdf, acessado em 05/05/2021, às 18:40.

MATLOFF, N. - Programming on Parallel Machines, 2011, url: http://160592857366.free.fr/joe/ebooks/ShareData/Programming%20on%20Parallel%20Machines.pdf, acessado em 15/05/2021, ás 10:00.

NVIDIA DEVELOPER - Cuda Zone - url: https://developer.nvidia.com/cuda-zone , acesso em 17/05/2021, às 19:00.

NVIDIA - CUDA C++ Programming Guide, 2021 - disponível em : https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, acesso em 02/05/2021

SOYOTA, T. - GPU Parallel Program Development Using CUDA, 2018.

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

BAUMES, L. A.; KRUGER, F.; JIMENEZ, S.; COLLET, P. CORMA A. - Boosting theoretical zeolitic framework generation for the determination of new materials structures using GPU programming - url : https://pubs-rsc-

org.ez180.periodicos.capes.gov.br/en/content/articlelanding/2011/CP/c0cp02833a#!divAbstract, acesso em 30/05/2021 às 17:15.

NVIDIA - NVIDIA GeForce GT 1030 : Visão geral - disponível em: https://www.nvidia.com/pt-br/drivers/geforce-gt-1030/#pdpContent=0, acesso em 04/07/2021, às 15:12.

ZOTAC - ZOTAC GeForce GT 1030 2GB GDDR5 HDMI/VGA Low Profile : Especificações do produto - disponível em: https://www.zotac.com/br/product/graphics_card/zotac-geforce-gt-1030-2gb-gddr5-hdmi-vga-low-profile#spec, acesso em 04/07/2021, às 16:19.

VAIDYA, B. - Hands-On GPU-Accelerated Computer Vision with OpenCV and CUDA, 2018.

COOK, S. - CUDA Programming : A Developer's Guide to Parallel Computing with GPUs, 2013.

FARBER, R. - CUDA Application Design and Development, 2011.

GRAMA, Ananth; GUPTA, Anshul; KARYPIS, George; KUMAR, Vipin - Introduction to Parallel Computing, Second Edition, 2003.

SZWARCFITER, Jayme Luiz; MARKENZON, Lilian - Estrutura de Dados e Seus Algoritmos, 2010, 3° edição, reimpressão 2015.

NVIDIA DEVELOPER - CUDA Dynamic Parallelism API and Principles - disponível em: https://developer.nvidia.com/blog/cuda-dynamic-parallelism-api-principles/, acesso em 25/09/2021, ás 18:59.

NVIDIA DEVELOPER - CUDA Toolkit documentation : CUDA Runtime API, CUDA Thrust API - disponível em: https://docs.nvidia.com/cuda/cuda-runtime-api/index.html, acesso em 25/09/2021, ás 19:23.

NVIDIA - Thrust documentation - disponível em : https://thrust.github.io/doc/index.html, acesso em 01/10/2021, às 20:53.

MALANÍK, David - Speed Differences Between Mathematica And C# In Search Of Extremes – disponível em : Trilobit - Speed differences between Mathematica and C# in search of extremes (utb.cz), acesso em 03/10/21, ás 16:16.

APÊNDICE A - PREPARAÇÃO DO AMBIENTE CUDA

Para preparar o ambiente de desenvolvimento, é preciso a instalação de alguns compiladores, drivers e outros componentes específicos ao uso de CUDA, o CUDA Toolkit, por exemplo. A instalação de CUDA envolve muitos passos, mas existe um caminho mais curto que a NVIDIA disponibilizou. Será usado aqui o caminho mais simples, mas para quem gosta de uma coisa mais elaborada com mais flexibilidade, é só acessar esse link: Guia de instalação de CUDA .

A.1 Instalação do compilador da linguagem C, GCC.

A seguir são mostrados os passos para a instalação do GCC nas plataformas Windows e Linux.

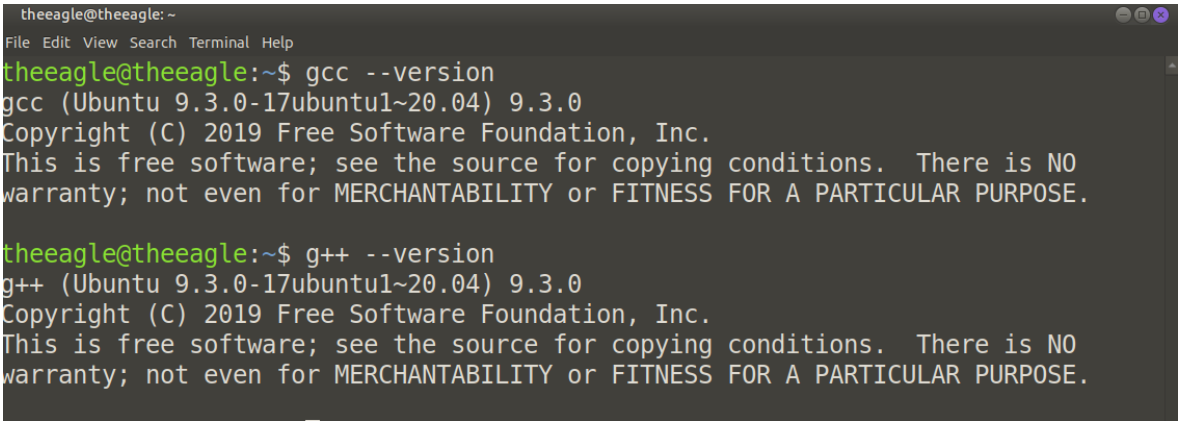
A.1.1 Instalação GCC no Linux [Ubuntu]

Para instalar o compilador GCC, é preciso executar alguns comandos. Antes de tudo, pode-se verificar que o mesmo não se encontra instalado, usando os comandos do item 4. Se não for o caso, abra um terminal Linux apertando **CTRL + ALT + T**, ou simplesmente, aperte a tecla com o ícone do windows e escreva **Terminal** e abrí-lo apertando **ENTER**. E, siga os passos a seguir, executando os comandos:

1. Atualize o sistema de pacote executando:
   `sudo apt update`
2. Instalar o compilador usando:
   `sudo apt install build-essential`
3. Instalar o manual para GCC usando (opcional):
   `sudo apt-get install manpages-dev`
4. Verifique a instalação usando os comandos:
   `gcc --version e g++ --version`
   gcc para verificar o compilador C e g++ para C++. Observe a figura 21.

Pode-se consultar esse artigo Install GCC Compiler on Ubuntu ou o fórum do site oficial cplusplus.com para mais informações.

Figura 22: Verificando a instalação de GCC e G++



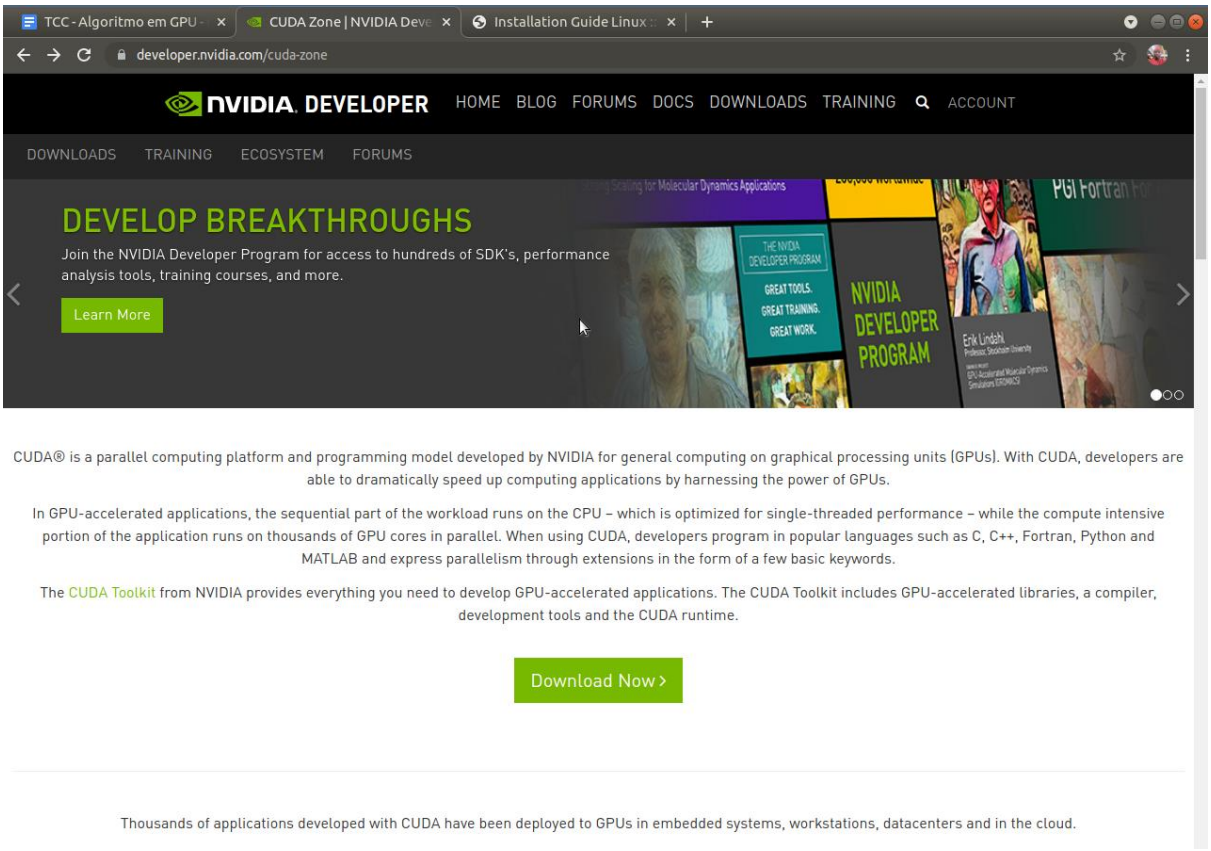
A.1.2 Instalação do GCC no Windows

A instalação do GCC no windows é geralmente feita junto a um IDE por exemplo CodeBlocks, acesse o link a seguir para baixá-lo, Download CodeBlocks for Windows. Para uma instalação mais customizada (instalar apenas o compilador), pode-se acessar o seguinte link Install GCC in Windows para instalá-lo.

A.2 Instalação de CUDA

Acessando esse link CUDA zone, será aberta a página oficial do site da NVIDIA tratando de CUDA, onde aparecerá um botão permitindo o download e instalação de CUDA; ou pode-se acessar direto a página de download pelo link Download CUDA. Clicando no botão, chegará na página de download. Estando lá, deve-se seguir os passos abaixo conforme o seu sistema operacional. Observe a figura a seguir:

Figura 23: Página CUDA zone da NVIDIA



A.2.1 Instalação no LINUX distribuição UBUNTU

1. Na próxima página, escolhe o sistema operacional, no caso **Linux**
2. Em seguida, escolhe a arquitetura. Para nosso estudo, escolhe **x86_64**
3. Agora, escolhe a distribuição, no nosso caso **Ubuntu**
4. A versão é **20.4**
5. Escolhe, agora a forma como quer instalar, **deb[local]** é largamente suficiente para nosso estudo.
6. Em seguida, uma lista de linhas de comandos aparecerá. Agora, basta seguir executando cada um na ordem para ter o CUDA instalado. Veja as figuras 24 e 25.
7. Após a instalação, deve-se acrescentar uma variável de ambiente; para isso, executar esse comando:

`<editor> ~/.profile`

será aberto um arquivo, nele, no final, acrescente essa linha:

`export PATH=/usr/local/cuda-<version>/bin${PATH:+:${PATH}}`

Substitua `<editor>` pelo editor de texto em que quer abrir o arquivo, por exemplo, code `~/.profile` para abrí-lo com Visual Studio Code (mais conhecido como VS code) e substitua `<version>` pela versão de CUDA, por exemplo, `cuda-11.4` . Pode-se acessar CUDA Post-installation Actions para mais informações.

8. Verifique a instalação com o comando abaixo:

   `nvcc --version`

   a saída deve ser semelhante à da figura 26.

Figura 24: Seleção das configurações para instalar CUDA

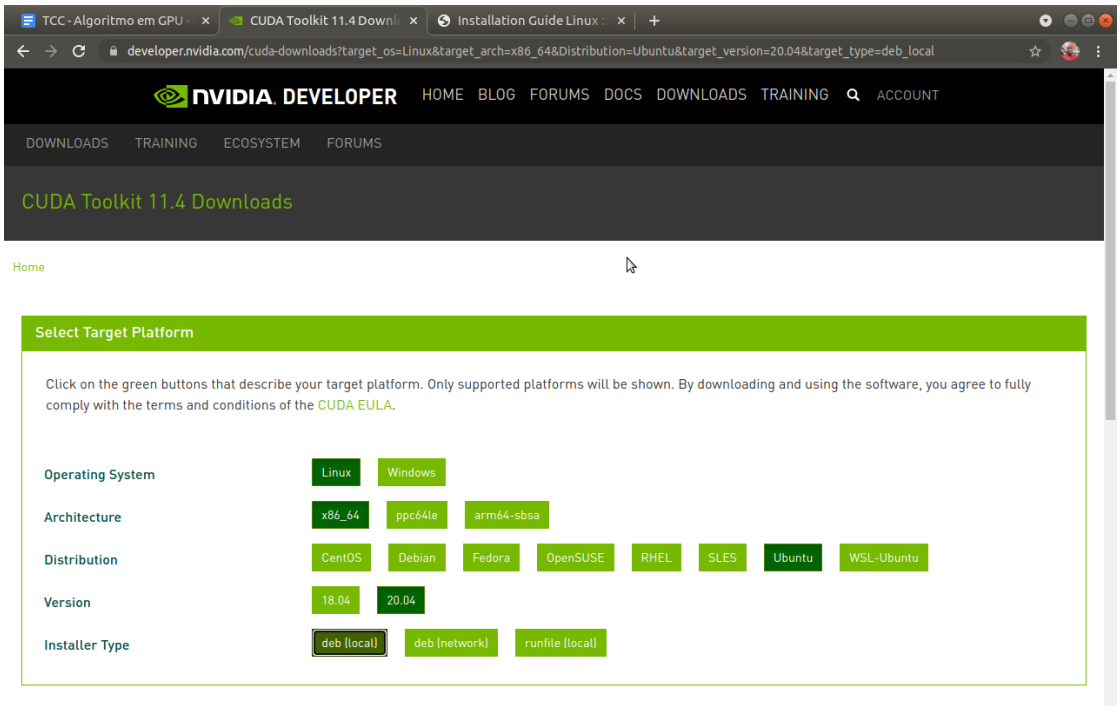


Figura 25: Linhas de comandos para a instalação

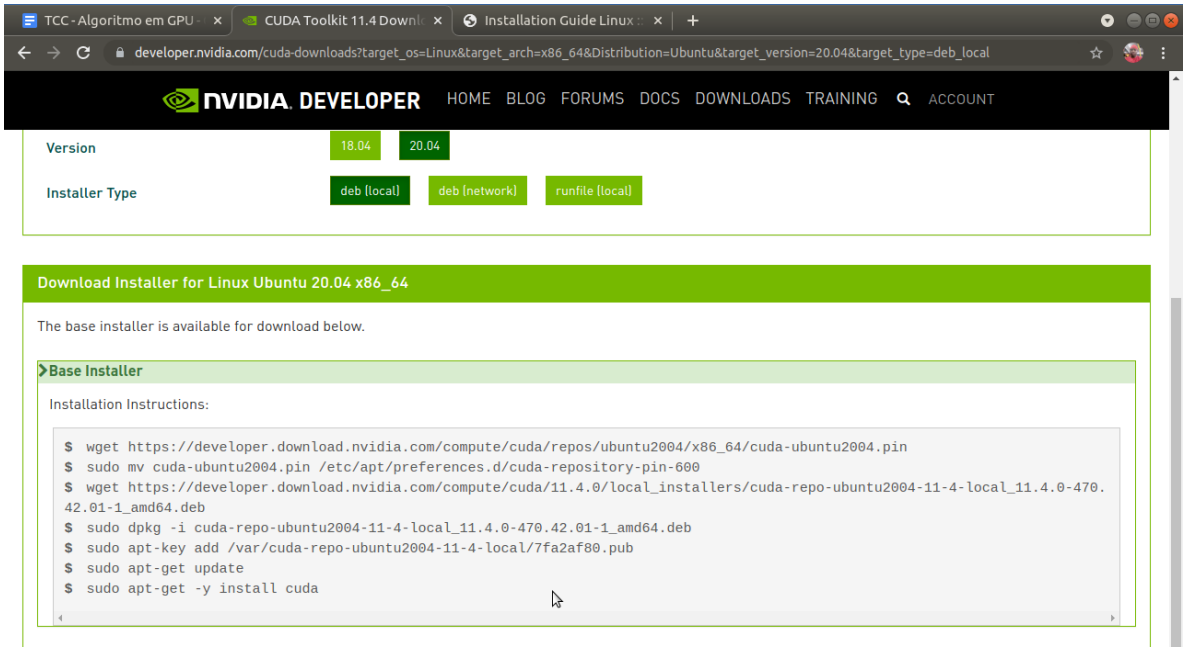
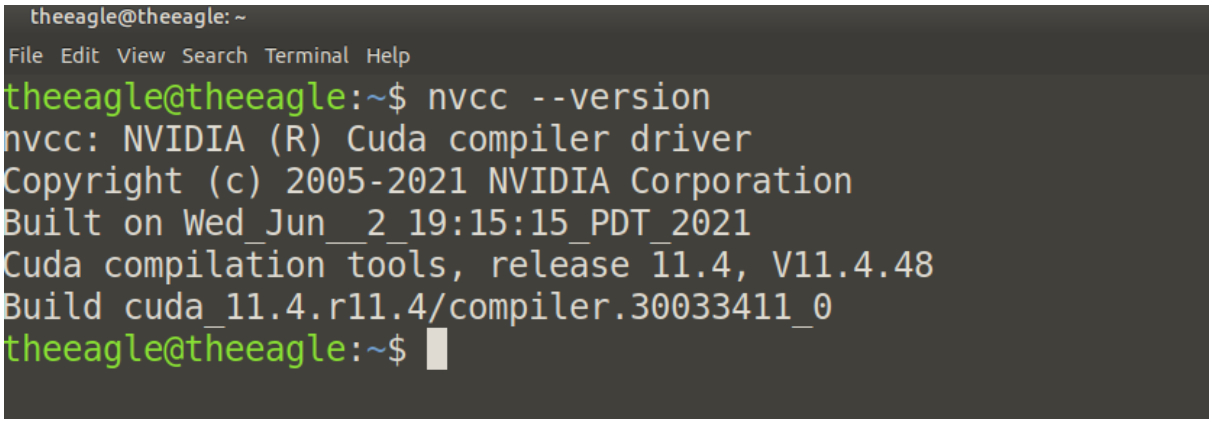


Figura 26: Saída na verificação da instalação de CUDA



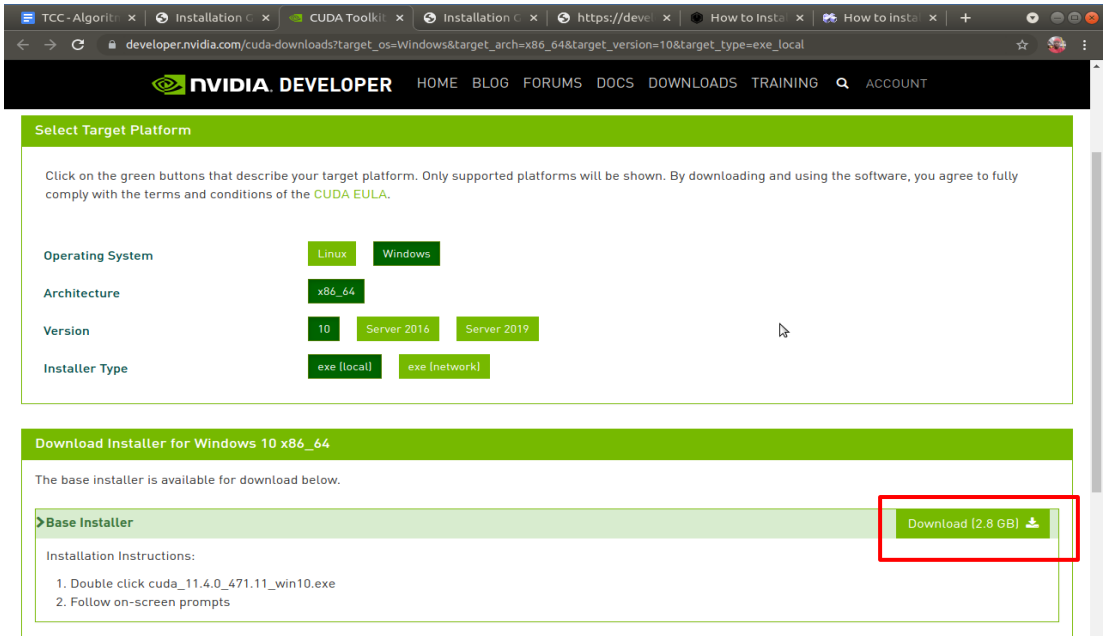
A.2.2 Instalação de CUDA na plataforma Windows

Para o sistema operacional Windows, os primeiros passos até a página de download são os mesmos. Na página de download siga os passos a seguir.

1. Escolha **Windows**
2. Escolha a arquitetura **x86_64**
3. Em seguida, a versão do Windows **10**, por exemplo.
4. Tipo de instalador, escolhe **exe[local]**
5. Download o setup (arquivo .exe), clicando no botão **Download (*tamanho* GB)**. Veja a figura 27.
6. Escolha a pasta de destino, e espere que acabe. Quando terminar o download, faça um **double click** sobre o setup, ou **um click a direita** e clique em **instalar** ou **abrir como administrador**. A partir daí, é só seguir os passos de uma instalação padrão do windows.
7. Verifique que a instalação foi bem sucedida seguindo os passos a seguir:
    a. Abra um **prompt** do windows apertando a tecla com o ícone do windows **+ R**, e escreva **cmd** e **ENTER**. No **prompt**, execute o comando: `nvcc --version` ou `nvcc -V`. A saída deve ser semelhante à saída da figura 26.
    b. Outra forma é verificar através das pastas, seguindo esse caminho: `C:\ProgramData\NVIDIACorporation\CUDA` Samples \v<version>\bin\win64\Release

s

Figura 27: Processo de instalação de CUDA em Windows



APÊNDICE B – CÓDIGO IMPLEMENTADO

B.1 Makefile

Como mencionado anteriormente, para kernel recursivo, algumas configurações a mais são preciso na etapa de compilação, isto é, alguns flags. Por isso, foi usado um `makefile` cujo conteúdo é o seguinte:

```
//makefile
program:
    nvcc -arch=sm_35 -rdc=true -lcudadevrt -Wno-deprecated-gpu-
targets -o main main.cu functions.cu

clean:
    rm main
```

Para compilar e executar, assegure-se de que todos os arquivos `(makefile, global.h, functions.cu, main.cu)` estejam na mesma pasta. Em seguida, use o comando `cd` via o Terminal para render-se até a pasta. E execute os comandos abaixos:

`$ make` para compilar

`$ ./main algoritmo numeroDeChaves` para executar.

Por exemplo, para uma execução com Insertion sort e 128 chaves, após compilar usando `make`, execute `./main 1 128`. Note que os algoritmos seguem a ordem de identificação seguinte: [1] Insertion sort [2] Ordenação por mesclagem (Merge) [3] Quicksort [4] Mergesort [5] CUDA thrust::sort.

B.2 global.h

O cabeçalho `global.h` contém os protótipos das funções implementadas

```
//global.h

#include<iostream>
#include<thrust/device_vector.h>
#include<thrust/host_vector.h>
#include <thrust/copy.h>
#include<thrust/sequence.h>
#include<cuda.h>
#include<cuda_runtime_api.h>
#include<cuda_runtime.h>
#include<time.h>
#include<stdlib.h>

#define MAX_RECURSION 24
#define WARP_SIZE 32

__global__ void vectorPrinter(int *, int, int);
__device__ void swapKernel(int *, int *);

__device__ void pivotMedianOf3Kernel(int [], long, long, long *);
__device__ void oddEvenSortKernel(int *, long, long);

__global__ void intercalateKernel(int *, int *, long, long, long);
__global__ void indexSearchKernel(int *, int, int, int *);
__global__ void pushElementsKernel(int *, int, int);

__global__ void rankCalculatorKernel(int *, int *, int *, int, int);
__global__ void rankSumCalculatorKernel(int *, int *, int);

__global__ void quickSortKernel(int *, long, long);
__global__ void mergeSortKernel(int *, int *, long, long);
__global__ void insertionSortKernel(int *, long, int *);
__global__ void mergeSortKernel(int *, int *, int *,  int *, int *, int
, int);

void mergeSort(int *, int *, int *, int, int);

void thrustSort(thrust::device_vector<int>&, double *);
```

B.3 Implementação das funções

```
//functions.cu

#include<thrust/device_vector.h>
#include<cuda_runtime_api.h>
#include<cuda_runtime.h>
#include<stdio.h>
#include<thrust/execution_policy.h>
#include <cub/cub.cuh>
#include <thrust/copy.h>

__device__ int MAX_RECURSION = 24;
__device__ int WARP_SIZE = 32;

__global__ void vectorPrinter(int *vector, int init, int nElements)
{
    for(int i = init; i < nElements; i++)
        printf("%d%s", vector[i], ((i > 0 && ((i % 20) == 0)) ? "\n" :
" "));
    printf("\n");
}

__device__ void swapKernel(int *firstKey, int *secondKey)
{
    int key = *firstKey;
    *firstKey = *secondKey;
    *secondKey = key;
}




__device__ void oddEvenSortKernel(int *vector,long init,long nElements)
{
    int target = 0;
    __shared__ int swaped;

    do
    {
        swaped = 0;

        for(int index = (init + target); index < (nElements - target);
                                                        index += 2)
            if(vector[index] > vector[index + 1]){
                swapKernel(&vector[index], &vector[index + 1]);
                 swaped += 1;
            }

        target = (target > 0) ? 0 : 1;
    } while (__syncthreads_count(swaped) != 0 ||  __syncthreads_count(t
                                                        arget) != 0);
}
```

```
__device__ void pivotMedianOf3(int vector[], long firstIndex, long last
Index, long *pivotIndex)
{
    long index = (firstIndex + lastIndex) / 2;
    if(vector[firstIndex] > vector[index])
    {   if(vector[firstIndex] < vector[lastIndex])
            *pivotIndex = firstIndex;
        else
        {
            if(vector[index] < vector[lastIndex])
                *pivotIndex = lastIndex;
            else
                *pivotIndex = index;
        }
    }else{
        if(vector[index] < vector[lastIndex])
            *pivotIndex = index;
        else
        {
            if(vector[firstIndex] < vector[lastIndex])
                *pivotIndex = lastIndex;
            else
                *pivotIndex = firstIndex;
        }
    }
}



__global__ void quickSortKernel(int *vector, long firstIndex, long last
Index)
{
    long pivotIndex;
    long firstTempIndex, lastTempIndex;
    int pivot;
    cudaStream_t leftStream, rightStream;

    if(((lastIndex - firstIndex) <= WARP_SIZE)){
        oddEvenSortKernel(vector, firstIndex, lastIndex+1);
        return;
    }
    else{
        pivotMedianOf3(vector, firstIndex, lastIndex, &pivotIndex);
        swapKernel(&vector[pivotIndex], &vector[lastIndex]);
        firstTempIndex = firstIndex;
        lastTempIndex = lastIndex - 1;
        pivot = vector[lastIndex];

        while(lastTempIndex >= firstTempIndex)
        {
```

```
            while(vector[firstTempIndex] < pivot)
                firstTempIndex++;

            while(vector[lastTempIndex] > pivot)
                lastTempIndex--;

            if(lastTempIndex >= firstTempIndex)
            {
                swapKernel(&vector[firstTempIndex], &vector[lastTempInd
                 ex]);
                firstTempIndex++;
                lastTempIndex--;
            }

        }
        swapKernel(&vector[firstTempIndex], &vector[lastIndex]);

        cudaStreamCreateWithFlags(&leftStream, cudaStreamDefault);
        quickSortKernel<<<1, 1, 0, leftStream>>>(vector, firstIndex, fi
            rstTempIndex-1);
        cudaStreamDestroy(leftStream);

        cudaStreamCreateWithFlags(&rightStream, cudaStreamDefault);
        quickSortKernel<<<1, 1, 0, rightStream>>>(vector, firstTempInde
            x, lastIndex);
        cudaStreamDestroy(rightStream);
    }
}

__global__ void intercalate(int *vector, int *tempVector, long firstInd
ex, long middle, long lastIndex)
{
    long leftPartIndex = firstIndex;
    long rightPartIndex = middle;
    long index = firstIndex;

    while((leftPartIndex < middle) && (rightPartIndex <= lastIndex))
    {
        if(vector[leftPartIndex] < vector[rightPartIndex])
        tempVector[index] = vector[leftPartIndex], leftPartIndex += 1;
        else
        tempVector[index] = vector[rightPartIndex],rightPartIndex += 1;
        index += 1;
    }

    while(leftPartIndex < middle)
        tempVector[index] = vector[leftPartIndex], leftPartIndex += 1,
            index += 1;

    while(rightPartIndex <= lastIndex)
```

```
        tempVector[index] = vector[rightPartIndex], rightPartIndex += 1
            , index += 1;

    cudaDeviceSynchronize();

    for(int i = firstIndex; i < lastIndex+1; i++)
        vector[i] = tempVector[i];

}


__global__ void mergeSortKernel(int *vector, int *tempVector, long firs
tIndex, long lastIndex)
{
    int middle;
    cudaStream_t leftStream, rightStream;

    if((lastIndex - firstIndex) > WARP_SIZE)
    {
        middle = (firstIndex + lastIndex) / 2;

        cudaStreamCreateWithFlags(&leftStream, cudaStreamDefault);
        mergeSortKernel<<<1, 1>>>(vector, tempVector, firstIndex, middl
            e);

        cudaStreamCreateWithFlags(&rightStream, cudaStreamDefault);
        mergeSortKernel<<<1, 1>>>(vector, tempVector, middle + 1, lastI
            ndex);
        cudaDeviceSynchronize();
        intercalate<<<1, 1>>>(vector, tempVector, firstIndex, middle +
            1, lastIndex);
        cudaStreamDestroy(leftStream);
        cudaStreamDestroy(rightStream);
    }
    else{
        oddEvenSortKernel(vector, firstIndex, lastIndex+1);
    }
}

__global__ void indexSearchKernel(int * vector, int limit, int number,
int * index)
{
    int threadId = (blockDim.x * blockIdx.x) + threadIdx.x;
    if(threadId < limit){
        if(vector[threadId] <= number && number < vector[threadId + 1])
            index[0] = threadId + 1;
    }
}

__global__ void pushElementsKernel(int * vector, int limit, int index)
{
```

```
    int threadId = (blockDim.x * blockIdx.x) + threadIdx.x;
    if(threadId >= index && threadId < limit){
        vector[threadId + 1] = vector[threadId];
    }
}

__global__ void insertionSortKernel(int * vector, long nElements, int *
index)
{
    __shared__ int number;
    int nThreadsPerBlock = 1024;
    int nBlocks = (nElements / nThreadsPerBlock) + 1;

    for(int i = 2; i < nElements+1; i++){

        index[0] = i;
        number = vector[i];

        indexSearchKernel<<<nBlocks, nThreadsPerBlock>>>(vector, i, num
           ber, index);
        cudaDeviceSynchronize();
        pushElementsKernel<<<nBlocks, nThreadsPerBlock>>>(vector, i, in
           dex[0]);
        cudaDeviceSynchronize();
        vector[index[0]] = number;
        cudaDeviceSynchronize();
    }
}

/**
*=====================================================================
* MergeSort, uma abordagem de baixo nível
* com paralelismo controlado
*=====================================================================
**/
__global__ void rankCalculatorKernel(int *a, int *b, int *rank, int na,
 int nb)
{
    int threadId = (blockDim.x * blockIdx.x) + threadIdx.x;
    int index = 0;

    if(threadId < na)
        while((index < nb) && (b[index] <= a[threadId]))
        {
            rank[threadId] += 1;
            index += 1;
        }
}

__global__ void rankSumCalculatorKernel(int *rankProper, int *rankCompa
red, int nElements)
```

```
{
    int threadId = (blockDim.x * blockIdx.x) + threadIdx.x;

    if(threadId < nElements)
        rankProper[threadId] += rankCompared[threadId];
}

__global__ void mergeSortKernel(int *a, int *b, int *c,  int *ra, int *
rb, int na, int nb)
{
    int threadId = (blockDim.x * blockIdx.x) + threadIdx.x;

    if(threadId < na)
        c[ra[threadId] - 1] = a[threadId];
    cudaDeviceSynchronize();

    if(threadId < nb)
        c[rb[threadId] - 1] = b[threadId];
    cudaDeviceSynchronize();
}

void mergeSort(int *a, int *b, int *c, int na, int nb)
{
    int nThreadPerBlock = 1024;
    int nABlocks = na / nThreadPerBlock;
    int nBBlocks = nb / nThreadPerBlock;
    int nCBlocks = (na + nb) / nThreadPerBlock;

    thrust::device_vector<int> rankAA(na);
    thrust::device_vector<int> rankAB(na);
    thrust::device_vector<int> rankBB(nb);
    thrust::device_vector<int> rankBA(nb);
    rankCalculatorKernel<<<nABlocks, nThreadPerBlock>>>(a, b, thrust::r
      aw_pointer_cast(&rankAB[0]), na, nb);
    rankCalculatorKernel<<<nBBlocks, nThreadPerBlock>>>(b, a, thrust::r
      aw_pointer_cast(&rankBA[0]), nb, na);
    rankCalculatorKernel<<<nABlocks, nThreadPerBlock>>>(a, a, thrust::r
      aw_pointer_cast(&rankAA[0]), na, na);
    rankCalculatorKernel<<<nBBlocks, nThreadPerBlock>>>(b, b, thrust::r
      aw_pointer_cast(&rankBB[0]), nb, nb);
    //cudaDeviceSynchronize();

    rankSumCalculatorKernel<<<nABlocks, nThreadPerBlock>>>(thrust::raw_
      pointer_cast(&rankAA[0]),
                                                           thrust::raw
      _pointer_cast(&rankAB[0]), na);
    //cudaDeviceSynchronize();
    rankSumCalculatorKernel<<<nBBlocks, nThreadPerBlock>>>(thrust::raw_
      pointer_cast(&rankBB[0]),
                                                           thrust::raw
      _pointer_cast(&rankBA[0]), nb);
```

```
    //cudaDeviceSynchronize();

    mergeSortKernel<<<nCBlocks, nThreadPerBlock>>>(a, b, c, thrust::raw
      _pointer_cast(&rankAA[0]),
                                                    thrust::raw_pointer
      _cast(&rankBB[0]), na, nb);

}

void thrustSort(thrust::device_vector<int>deviceVector, float *executed
Time)
{
    cudaEvent_t startTime, endTime;

    cudaEventCreate(&startTime);
    cudaEventCreate(&endTime);

    cudaEventRecord(startTime, 0);

    thrust::sort(deviceVector.begin(), deviceVector.end());

    cudaEventRecord(endTime, 0);

    cudaEventSynchronize(endTime);

    cudaEventElapsedTime(executedTime, startTime, endTime);

}
```

B.4 Main para testes

```
//main.cu

#include "global.h"
#include<sys/time.h>
#include<stdint.h>
using namespace std;

int32_t nElements;

int _rand()
{
    return rand() % nElements;
}

int main(int argc, char **argv)
{
    int algorithm = atoi(argv[1]);
    nElements = atoi(argv[2]);
```

```
    struct timeval startTime, endTime;
    double executedTime;

    srand(time(NULL));
    //vetor principal da ordenação
    thrust::device_vector<int> deviceVector(nElements);

    cudaDeviceSetLimit(cudaLimitDevRuntimeSyncDepth, MAX_RECURSION);

      if(algorithm == 1)
    {
        cout << " [1] Insertion Sort:" << endl;

        thrust::device_vector<int>index(1);
        thrust::host_vector<int> hostVector(nElements);

        deviceVector.resize(nElements + 2);
        deviceVector[0] = -99999999;
        deviceVector[nElements + 1] = 99999999;

        thrust::generate(hostVector.begin(), hostVector.end(), _rand);
        thrust::copy(hostVector.begin(), hostVector.end(), deviceVector
            .begin() + 1);
        cudaDeviceSynchronize();

        gettimeofday(&startTime, 0);
        insertionSortKernel<<<1, 1>>>(thrust::raw_pointer_cast(&device
            ector[0]),
        nElements, thrust::raw_pointer_cast(&index[0]));
        cudaThreadSynchronize();
        gettimeofday(&endTime, 0);
    }else if(algorithm == 2){
        cout << "[2] Merge Sort:" << endl;

        thrust::device_vector<int> A(nElements);
        thrust::device_vector<int> B(nElements);

        thrust::sequence(A.begin(), A.end(), 0, 3);
        thrust::sequence(B.begin(), B.end(), 1, 3);
        deviceVector.resize(2 * nElements);

        cudaDeviceSynchronize();

        gettimeofday(&startTime, 0);

        mergeSort(thrust::raw_pointer_cast(&A[0]),
        thrust::raw_pointer_cast(&B[0]),
        thrust::raw_pointer_cast(&deviceVector[0]),
        nElements, nElements);
```

```
        cudaThreadSynchronize();
        gettimeofday(&endTime, 0);

    }else if(algorithm == 3){
        cout << "[3] Quicksort:" << endl;

        thrust::host_vector<int> hostVector(nElements);

        thrust::generate(hostVector.begin(), hostVector.end(), _rand);
        thrust::copy(hostVector.begin(), hostVector.end(), deviceVector
            .begin());
        cudaDeviceSynchronize();

        gettimeofday(&startTime, 0);

        quickSortKernel<<<1, 1>>>(thrust::raw_pointer_cast(&deviceVecto
            r[0]),
        0, nElements - 1);
        cudaThreadSynchronize();
        gettimeofday(&endTime, 0);
    }
    else if(algorithm == 4){
        cout << "[4] Mergesort recursivo:" << endl;

        thrust::device_vector<int> tempVector(nElements);
        thrust::host_vector<int> hostVector(nElements);

        thrust::generate(hostVector.begin(), hostVector.end(), _rand);
        thrust::copy(hostVector.begin(), hostVector.end(), deviceVector
            .begin());
        cudaDeviceSynchronize();

        gettimeofday(&startTime, 0);

        mergeSortKernel<<<1, 1>>>(thrust::raw_pointer_cast(&deviceVecto
            r[0]),
                                  thrust::raw_pointer_cast(&tempVecto
            r[0]),
                                  0, nElements - 1);
        cudaThreadSynchronize();
        gettimeofday(&endTime, 0);
    }
    else{
        puts("thrust::sort :");
        //float executedTimeThrustSort;
        thrust::host_vector<int> hostVector(nElements);

        thrust::generate(hostVector.begin(), hostVector.end(), _rand);
```

```
        thrust::copy(hostVector.begin(), hostVector.end(), deviceVector
            .begin());
        cudaDeviceSynchronize();

        gettimeofday(&startTime, 0);
        thrust::sort(deviceVector.begin(), deviceVector.end());
        //thrustSort(deviceVector, &executedTimeThrustSort);
        //printf("\nCom thrustSort, a ordenação gastou : %f milisegundo
            s\n", executedTimeThrustSort);
        gettimeofday(&endTime, 0);
    }

    executedTime = endTime.tv_sec + (endTime.tv_usec / 1000000.0);
    executedTime = executedTime - (startTime.tv_sec +(startTime.tv_usec
      / 1000000.0));
    printf("\n A ordenação gastou : %f segundos\n", executedTime);
    vectorPrinter<<<1, 1>>>(thrust::raw_pointer_cast(&deviceVector[0]),
      s0, nElements);

    cudaDeviceSynchronize();

    return 0;
}
```