Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Caio Costa Salgado

Acelerando transporte radiativo ao redor de buracos negros com GPUs

São Paulo
Dezembro de 2017

Uso de GPGPU na Análise de Buracos Negros

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Rodrigo Nemmen da Silva

São Paulo
Dezembro de 2017

Resumo

Aqui vai o resumo que ainda tem que ser feito....

Palavras-chave: GPGPU, CUDA, HPC, Monte Carlo, Transferência radioativa, Buraco Negro.

Abstract

And here will be the english abstract, that still need to be done....

Keywords: GPGPU, CUDA, HPC, Monte Carlo, Rasioactive Transfer, Black Hole.

Sumário

Lista de Abreviaturas

GPU	Unidade de processamento Gráfico (*Graphics Grocessing Unit*)
CPU	Unidade de Processamento Central (*Central Processing Unit*)
GPGPU	Unidade de processamento Gráfico de Propósito Geral (*General Purpose Graphics Processing Unit*)
CUDA	Computação em Arquitetura unificada de dispositivos (*Compute Unified Device Architecture*)
HCP	Computação de Alta Performance (*High Performance Computing*)
SIMD	Única Instrução Múltiplos dados (*Sigle Instruction Multiple Dada*)

Capítulo 1

Introdução

Uma grande dúvida dos astrofísicos e também de toda a comunidade científica é o que ocorre em um buraco negro e em suas proximidades. Na busca de respostas programas de computador são feitos com o intuito de simular essa região e talvez trazer alguma luz, um desses programas é o **grmonty**(Dolence *et al.*, 2009) (nome reduzido, em inglês, de Monte Carlo para Relatividade Geral).

Programas dessa natureza tendem a ser muito intensos no que diz respeito ao processamento, exigindo muito das CPUs, estas tornam-se assim um limitante, um gargalo, para a velocidade com a qual o programa pode devolver um resultado. É neste contexto que buscamos aplicar métodos de *Computação de Alta Performance* para otimizar ao máximo o uso todos os dispositivos do computador (hardware) que temos disponíveis.

Muitas das técnicas de HPC exploram a paralelização, o que pode ser feito massivamente por um hardware específico nesse caso as *unidades de processamento gráfico*, GPU. Tais dispositivos são muito populares e já presentes em muitas máquinas domésticas e até em smartphones, eles são confeccionados primordialmente para processamento gráfico em jogos digitais, porém graças aos avanços recentes tais dispositivos tem se tornado mais genéricos e respondendo a uma gama maior de problemas.

Ao analisar o funcionamento do **grmonty** - por sua característica de simulador de partículas - é possível classificar parte de sua execução no modelo SIMD (sigla em inglês para única instrução múltiplos dados), uma vez que simula a trajetória da cada fóton de maneira independente. Dada essa informação podemos explorar o poder computacional das GPUs afim de paralelizar a execução do código, aumentando drasticamente sua a performance.

Existem vários programas que simulam essas regiões próximas a buracos negros, porém poucos tem a amplitude e relacionam diferentes propriedades físicas, em especial a relatividade geral, como o **grmonty**. Tornando o programa mais performático e lhe fornecendo a capacidade de executar em computadores domésticos com dados e precisão relevantes, prestaria um grande avanço na pesquisa de buracos negros, facilitando a execução de simulações e dimanado o tempo de espera por dados.

O **grmonty** também apresenta algumas outras características que o tornam mais atra-

ente no quesito de programa que pode ser otimizado. Como ele é relativamente pequeno (menos de 5 mil linhas), possuir a característica de ter uma arquitetura SIMD e ser feito todo em linguagem C, uma portabilidade para ser executado em GPUs é um ato factível e que pode apresentar grandes ganhos com um esforço não muito alto, aproveitando assim da paralelização massiva que as placas gráficas apresentam.

Este trabalho tem como objetivo apresentar melhorias a execução do código do **grmonty**, utilizando-se do processador de placas gráficas, as GPUs, para massivamente paralelizar e distribuir a carga de trabalho pelos múltiplos núcleos de processamento destas placas. Primeiramente é explicado o que é o **grmonty** e como funciona, depois os paradigmas ao qual sua execução apoia-se, caminhando para a explicação de GPUs e como contribuem para o aumento de performance, assim são apresentadas as otimizações executadas, chegando finalmente nos resultados alcançados e conclusões. Há ainda um capítulo de próximos passos demonstrando que ainda há muito espaço para mais melhorias e mais velocidade na execução.

Capítulo 2

Grmonty: Monte Carlo para Relatividade Geral

2.1 O que Faz

Dolence et al definem o **grmonty** como "software destinado a calcular o espectro de plasmas quentes e opticamente finos a par da completa relatividade geral utilizando um código de transporte radioativo baseado na técnica de Monte Carlo"(Dolence *et al.*, 2009, p.1, traduzido). Em outras palavras o programa estima o espectro de uma simulação de magnetoidrodinâmica Alfvén (1942) relativística utilizando o método de Monte Carlo.

Utilizando o método de Monte Carlo na geração dos dados, os fótons, e a partir de um dado modelo de plasma fornecido como entrada, o qual especifica velocidade, densidade, força do campo magnético e temperatura, o programa busca gerar o espectrograma. Para tanto um número próximo a N - fornecido na entrada - de fótons é gerado e para cada fóton sua trajetória é traçada. Nessa trajetória o fóton é espalhado e pode passar por diferentes interações, até finalmente ser mensurado.

Depois de algumas iterações um número próximo a N de fótons já foi gerado e rastreado, assim um relatório com o espectrograma é obtido e retornado pelo programa que finalmente termina.

2.2 Para que Faz

Foram desenvolvidas várias técnicas para calcular a transferência radioativa a partir de fontes como as descritas a baixo(Dolence *et al.*, 2009), porém poucas levam em conta a relatividade geral como um todo, principalmente no quesito de objetos, fontes, muito massivas ou com velocidades próxima a da luz, o **grmonty** vem para aprimorar esses cálculos.

Qualquer fonte astrofísica de radiação que seja relativística, ou seja, qualquer corpo ou fenômeno fonte de radiação eletromagnética, seja do rádio à raios gama e que é relativística: apresenta uma considerável distorção no espaço-tempo, seja por estar em velocidades próxi-

mas a da luz, seja por possuir uma enorme quantidade de massa e/ou energia. Exemplos de objetos são os buracos negros e estrelas de nêutrons, fenômenos são os *Gamma Ray Bursts* ou núcleos ativos de galáxias.

2.3 Como Faz

No momento de criação e rastreio dos fótons o programa faz o uso da biblioteca **OpenMP** para paralelizar o desenvolvimento dos fótons, graças a esta abordagem é viável o potencial uso de todos os núcleos disponíveis na CPU da máquina. A biblioteca é utilizada para que cada fóton seja produzido e espalhado de forma independente dos outros e funcionando em paralelo, além disso todas as instruções não dependem do fóton em si, elas são as mesmas instruções para todos os fótons. Desta forma podemos caracterizar o **grmonty** como tendo uma computação SIMD.

"Única Instrução Múltiplos Dados: Nesse tipo de computação podem haver múltiplos processadores, cada um operando sobre seu item de dados, mas estão todos executando a mesma instrução naquele item de dados"(Eijkhout *et al.*, 2016, p.84, traduzido). A arquitetura SIMD trabalha em ressonância com o **OpenMP** uma vez que torna a paralelização muito simples de ser aplicada: não há variáveis compartilhadas, não há condicionais ou desvios de fluxo que tornem cada execução diferente uma da outra, não há necessidade de sincronização ou *mutex*. Tornar o programa paralelizável é simples já que requer um uso mínimo do ferramental de programação concorrente.

Toda a vez que um fóton é criado logo em seguida sua rota é traçada, a relação entre criação e cálculo de trajetória é de 1 para 1. O que é evidente ao se observar as linhas 106 a 137 do *grmonty.cu*, aqui copiadas:

```
1     #pragma omp parallel private(ph)
2     {
3       while (1) {
4         /* get pseudo-quanta */
5     #pragma omp critical (MAKE_SPHOT)
6         {
7           if (!quit_flag)
8             make_super_photon(&ph, &quit_flag);
9         }
10        if (quit_flag)
11          break;
12
13        /* push them around */
14        track_super_photon(&ph);
15
16        /* step */
```

```
17      #pragma omp atomic
18          N_superph_made += 1;
19          /*mais codigo*/
20        }
21      }
```

Fica claro também - ao observar a linha 8 a 14 - que o processamento do rastreio é feito assim que possível, ao oposto de um processamento em lotes, ou seja, assim que o comandado *make_super_photon* é executado, gerando um novo fóton *ph*, o *track_super_photon* é chamado, não havendo algum buffer ou lote, um fóton produzido é um fóton consumido.

Tal processamento reduz muito os vestígios que um fóton pode criar durante sua existência. Uma vez que não se perde tempo deixando-o na memória, assim que é mensurado seu espaço na memória já é ocupado pelo próximo fóton a ser produzido, há um foco na economia de memória. O número de fótons na memória é o número de threads rodando simultaneamente.

Por fim se faz necessário notar que o programa é escrito na linguagem de programação C. O que faz muito sentido do ponto de vista de performance, uma vez que C é uma linguagem de baixo nível, mais próxima a linguagem de máquina e por isso é quase sempre explícito a quantidade e de que forma se está manipulando a memória. Outras vantagens são as possibilidades de usar tanto a biblioteca **OpenMP** como as otimizações do **gcc**, o *Gnu C Compiler*, mas do ponto de vista da expressividade uma linguagem de mais alto nível poderia apresentar outras vantagens, como uma maior legibilidade do código e o uso de abstrações e encapsulamento, aumentando também a capacidade e a facilidade de fazer manutenções e melhorias no código.

Capítulo 3

GPGPU

A seguir é explanado o que vem a ser uma placa gráfica, principalmente seu processador, a GPU. Depois é descrita a motivação para a qual esse hardware foi criado, o por quê ainda é produzido e por que provavelmente continuará. Seguindo são apontadas as diferenças arquiteturais entre uma CPU e uma GPU, vantagens e desvatagens. Na continuação são apresentadas e demonstradas as duas principais linguagens que permitem o acesso a programação genérica na GPGPU a OpenCL e a CUDA, contendo o por quê neste projeto foi escolhido a linguagem CUDA. Por fim são apresentadas aplicações que se utilizam do estado da arte no hardware e software das placas gráficas mais atuais e avançadas.

3.1 Placas de vídeo

Placa de vídeo é uma peça que pode ou não estar presente em um computador, ela é responsável por fornecer o hardware especializado em renderização gráfica, projetado para prover um grande aumento de performance a um baixo custo se comparado a CPUs no que disrespeito a tranformação de dados na memoria em imagens e vídeos, prontos para serem apresentados em telas e monitores. São constituídas principalmente de:

- Um chip de memória, cuja a principal função é armazenar texturas, ou qualquer outro dado que deverá ser várias vezes utilizado, consultado, no processo de renderização.
- Portas de acesso e conexão a monitores e telas, variam entre VGA (Video Graphics Array), DVI (Digital Visual Interface) e HDMI (High-Definition Muiltimidia Interface). São as saídas mais comuns às computações das placas.
- Um processador, uma GPGPU composta de alguns milhares de núcleos
- Uma ou várias ventoinhas, para dissipar o calor produzido pelo processador

[1]Disponível em https://commons.wikimedia.org/wiki/File:NVIDIA-GTX-1070-FoundersEdition-FL.jpg Domínio público

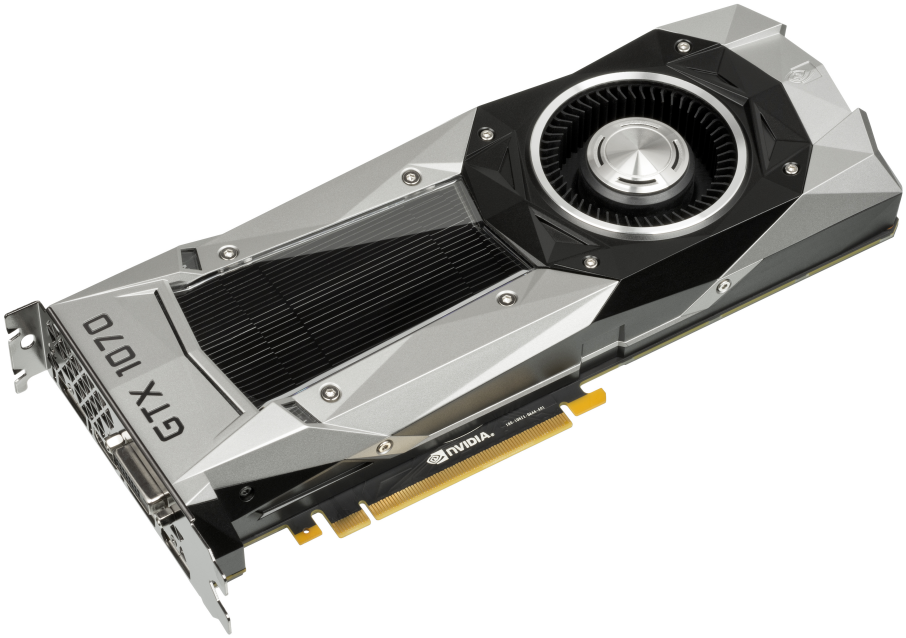

Figura 3.1: *Placa grafica NVIDIA GTX 1070 para computadores de mesa "desktop", essa versão é a "Founders edition", possui cerca de 8 GBs de memória com uma banda de 256 GB/s , 1920 núcleos, frequência de funcionamento de 1506 MHz e consome 150W. (Fonte: Wikimedia Commons [1])*

A figura 3.1 é uma foto de uma placa gráfica da Nvidia, a NVIDIA GTX 1070, começou a ser comercializada em Junho de 2016 à um preço de 379 dólares e possuí 1920 núcleos de processamento à 1,506 GHz em sua GPU (Balraj, 2016). Nessa mesma época também é comercializada o *Intel® Core™ i7-6850K Processor* CPU da Intel, vendida na época a partir de 617 dólares, possui 6 núcleos à uma frequência de 3,6 GHz (Intel, 2016). Um cálculo simples de tiques por segundo, ou seja, quantidade de ciclos que podem ser executados por segundo para cada processador demonstra que, enquanto a CPU da intel é capaz de realizar 21,6 bilhões de ciclos, no total, por segundo a GPGPU da nvidia é capaz de 2891,52 bilhões de ciclos, no total, por segundo, uma diferença de aproximadamente 133,8 vezes. A GPU que é 60,42% do valor da CPU é mais de 100 vezes mais rápida.

GPGPU é uma acrônimo para *General Purpose Graphics Processing Unit* em tradução literal: Unidade de processamento Gráfico de Propósito Geral. É uma evolução, uma adaptação que as GPUs passaram, na qual sua cadeia de processamento gráfico foi flexibilizada tornado possível usá-la para propósitos mais gerais, indo muito além do escopo de produção de gráficos e imagens em três dimensões, porém suas origens e modo de operação advem da sua função original de processar uma cadeia de dados gráficos, e tal necessidade determinou qual seria o objetivo da arquitetura que tais processadores deveriam ter. Assim entender essa cadeia de processamento ou *pipeline* dos dados das placas gráficas é importante para entender o por quê são como são (Kirk e mei Hwu, 2016).

O objetivo das GPUs é gerar imagens e vídeos que representam visões de uma cena virtual. A visão desta cena é descrita pela posição de uma câmera virtual e é definida pela geometria, orientação e propriedades materiais da superfície dos objetos na cena representados, bem como das propiedades das fontes de luz. APIs gráficas como OpenGL (The Khronos Group Inc., 2018a), DirectX (Microsoft, 2018) ou Vulkan(The Khronos Group Inc., 2018b) representam este processo como um pipeline que executa uma série de operações sobre um conjunto de vértices enviados pela CPU, sendo que cada vértice possui algumas propriedades, tais como cor, posição e vetor normal (Fatahalian e Houston, 2008).

O vertex shader é um programa que executa um conjunto de operações para cada um dos vertíces de entrada, com o intuíto de projetar cada vértice, baseado em sua posição relativa à camera virtual, em um espaço de tela 2D. Destes vértices, é montado um conjunto de triângulos, que representam os objetos no espaço 2D. Assim, quanto maior a quantidade de triângulos, melhor a qualidade com que o objeto será representado (Fatahalian e Houston, 2008). Visto que cada cena possui milhares de vértices e cada um deles pode ser tratado independentemente, os vértices podem ser processados paralelamente (GREEN *et al.*, 2008).

A seguir, ocorre o processo de rasterização, que consiste em determinar quais espaços da tela são cobertos por cada triângulo. Esse processo resulta na geração de fragmentos para cada espaço de tela coberto. Um fragmento é o que virá a ser um pixel, que contem todas as informações necessárias para gerar um na imagem final (profundidade, localização no frame buffer, etc.). A partir da posição da câmera virtual, os fragmentos que são ocultos por outros fragmentos são descartados, só é necessário mostras os objetos que podem ser vistos (GREEN *et al.*, 2008).

Já o pixel shader opera sobre a saída gerada pelo processo de rasterização. O pixel shader é um programa que consiste em um conjunto de operações que são executadas sobre cada fragmento, antes que estes sejam plotados na tela. Utilizando as informações de cor dos vértices e, possivelmente, buscando dados adicionais na memória global em forma de texturas (é nesse momento que a memória principal da placa é necessária), cada fragmento é processado para obter-se a cor final do pixel. Assim como na etapa de processamento de vértices, os fragmentos são independentes e podem ser processados em paralelo. Esta etapa é a que tipicamente demanda maior processamento dentro da estrutura do pipeline gráfico (GREEN *et al.*, 2008).

Uma vez que os programas de shader necessitam ser aplicados em milhares de vértices e pixels independentemente, as GPUs evoluíram para um conjunto de multiprocessadores massivamente paralelos. Além disso, dependendo do balanceamento da carga de trabalho da aplicação, apesar dos pixels serem dependentes dos vértices, ambos podem ser executados paralelamente. Esta característica resultou no aumento da programabilidade dos multiprocessadores da GPU (Kirk e mei Hwu, 2016).

Essa cadeia de processamento é o que destacou as GPUs das CPUs convencionais. No desenvolvimento de processadores dedicados a processar essa cadeia na forma mais eficiente possível as GPUs tomaram um caminho mais focado na paralelização, com mais núcleos,

mais unidades logico-aritméticas em detrerimento de mais ciclos por segundo, priorizando uma única memória grande e não muito rápida ao invés de varias camadas de memória cache muito rápida.

Esta arquiterura muito focada, ganhou a atenção de outras computações não necessariamente ligadas a renderização de imagens. Tal demanda eclodiu na flexibilização das GPUs para GPGPUs.

intel graphycs

3.2 A História das GPU e GPGPU

As GPUs, a princípio, não foram criadas para cálculos numéricos ou simulações de particulas e sim para a rederização de imagens para jogos digitais, o alto custo na produção e aquisição de hardware mais potente, pricipalmente a memória, na década de 70 impulsionou a criação de processadores mais dedicados e com funções mais específicas para a rederização gráfica. Impulsionada pelo mercado de jogos de video game, e buscando evitar o alto valor de chips de memória, placas de sistemas de arcade apresentavam uma composição de chips de vídeo para combinar os dados enquanto estes eram escaneados saindo para o monitor (Hague, 2013).

Em 1977 o Atari 2600 usou um *video shifter*, um tipo de circuíto integrado responsável por emitir o sinal de TV, chamado *Television Interface Adaptor* ou TIA. Foi customizado para ser capaz de gerar as imagens finais para a tela, os efeitos sonoros e ler os comandos vindos dos joysticks, teve como direcionamento em seu design a economia de memória RAM, muito cara na época (Hague, 2013) (Atari, 1983).

Em 1985 o Commodore Amiga apresentou um chip gráfico que vinha com um circuíto *blitter*, para maior velocidade na manipulação de memória, tranformação de bitmaps, desenho de linhas e funções de preenchimento de áreas. Também vinha com um co-pocessador, capaz de executar instruções únicas e manipular os registradores em sincronia com o canhão de desenho dos tubos de raios catódicos das televisões (Jessen, 2017).

Na decada de 90 a nintendo e a sony criaram seus consoles de vídeo game, o nintendo 64 e o playstation, ambos capazes de produzir gráficos em 3 dimensões a partir de polígonos. A distição entre os dois principais processadores desses vídeo games eram claras, a nintendo já havia detalhado que seu sistema vinha com dois processadores: um de propositos mais geral o *MIPS R4200* e o *Reality Coprocessor* desenhado para cálculos em 3d de alta performance, pre-processamento de aúdio e vídeo, mapeamento de textura e buferização de profundidade em tempo real (Nintendoi of America Inc., 1998) (McCall, 1996).

A nintendo não utilizou o termo CPU para descrever seu *Reality Coprocessor*, foi a Nvidia em 1999 que popularizou o termo. Ele já existia desde década de 80, porém somente ápos a capanha de marketing de sua placa de vídeo, a GeForce 256 com o logo: *"the world's first GPU"* (A primeira GPU do mundo) que o termo ficou mais comum (**?**).

Já nos anos 2000 teve mais NVIDIA e ATI By October 2002, with the introduction of the ATI Radeon 9700 (also known as R300), the world's first Direct3D 9.0 accelerator, pixel and vertex shaders could implement looping and lengthy floating point math, and were quickly becoming as flexible as CPUs, yet orders of magnitude faster for image-array operations. Pixel shading is often used for bump mapping, which adds texture, to make an object look shiny, dull, rough, or even round or extruded.[37]

Nos dia atuais: The PS4 and Xbox One were released in 2013, they both use GPUs based on AMD's Radeon HD 7850 and 7790. Nvidia's Kepler line of GPUs was followed by the Maxwell line, manufactured on the same process. 28 nm chips by Nvidia were manufactured by TSMC, the Taiwan Semiconductor Manufacturing Company, that was manufacturing using the 28 nm process at the time. Compared to the 40 nm technology from the past, this new manufacturing process allowed a 20 percent boost in performance while drawing less power.[48][49] Virtual reality headsets like the Oculus Rift and the HTC Vive have very high system requirements. Headset manufacturers have recommended GPUs for good virtual reality experiences. At their release, they had the GTX 970 from Nvidia and the R9 290 from AMD as the recommended GPUs.[50][51] Pascal is the newest generation of graphics cards by Nvidia released in 2016. The GeForce 10 series of cards are under this generation of graphics cards. They are made using the 16 nm manufacturing process which improves upon previous microarchitectures.[52] The Polaris 11 and Polaris 10 GPUs from AMD are made with a 14 nm process. Their release results in a big increase in the performance per watt of AMD video cards.[53]

3.3 CPU vs GPU

Em 2009, o poder computacional para operações de ponto flutuante de uma GPU era cerca de dez vezes superiores ao de uma CPU tradicional. Já a sua largura de banda era cerca de três vezes superior (KIRK; HWU, 2010). Para compreender os motivos desta diferença de desempenho, é necessário entender os propósitos de cada um desses processadores. A CPU é otimizada para desempenho de código sequencial, sendo que nessa, geralmente, utilizam-se de uma lógica de controle mais sofisticada, fazendo com que a CPU seja projetada com mecanismos como previsão de desvios e execução fora de ordem. Desta forma, permite-se que um único fluxo de instruções seja executado em paralelo enquanto a aparência de uma execução sequencial é mantida. Além disso,19 a CPU também é otimizada de forma a reduzir a latência de acesso aos dados e às instruções na memória principal. De fato, grande parte dos recursos da CPU são utilizadas para o gerenciamento de vários nıveis de memória cache, que possuem o objetivo de minimizar o tempo necessário para que os dados requisitados sejam de fato utilizados pelo processador. Uma vez que a CPU tem como alvo programas de propósito geral, poucos de seus recursos são utilizados para processamento de operações de ponto flutuante (KIRK; HWU, 2010). A GPU é adequada para a solução de problemas

onde o processamento dos dados pode ser realizado massivamente em paralelo, ou seja, o mesmo programa é executado simultaneamente em muitos elementos com alta intensidade aritmética (NVIDIA, 2010a). Por causa disto, as GPUs dedicam a maior parte de seus recursos ao processamento de dados ao invés de cache de dados e controle de fluxo (KIRK; HWU, 2010). Na Figura 2.1, pode-se observar a maneira com que os transistores do hardware são empregados nos chips da CPU e GPU. Na CPU, grande parte da área do chip é dedicada à memória cache e lógica de controle. Na GPU, diferentemente da CPU, a maior parte dos recursos do chip é projetada para processamento massivamente paralelo de operações de ponto-flutuante. Assim, sua capacidade de processamento é aumentada consideravelmente (NVIDIA, 2010a).

Na GPU, grande parte dos dados é utilizada uma única vez, sendo que estes dados precisam ser transferidos da memória principal da GPU para o uso de seus multiprocessadores. Desta forma, o sistema de memória da GPU é projetado para maximizar a taxa de transferência de dados (throughput), ou seja, a GPU é projetada de forma a aumentar a largura de banda, ao invés de minimizar a latência para o acesso aos dados. Esta grande largura de banda é obtida através do uso de amplos barramentos e memórias dynamic random access memory (DRAM) do tipo graphics double data rate (GDDR) (FATAHALIAN; HOUSTON, 2008). É importante destacar que nem todas as aplicações terão um desempenho melhor na GPU em relação à CPU. As GPUs são projetadas como engines de processamento20 numérico, e não executam bem os programas sequenciais que as CPUs normalmente executam. Assim, deve-se utilizar a placa GPU como um co-processador para au- xiliar a CPU na execução dos programas, de forma que sejam atribuıdas as partes sequenciais à CPU e as partes com processamento paralelo aritmético à GPU (KIRK; HWU, 2010).

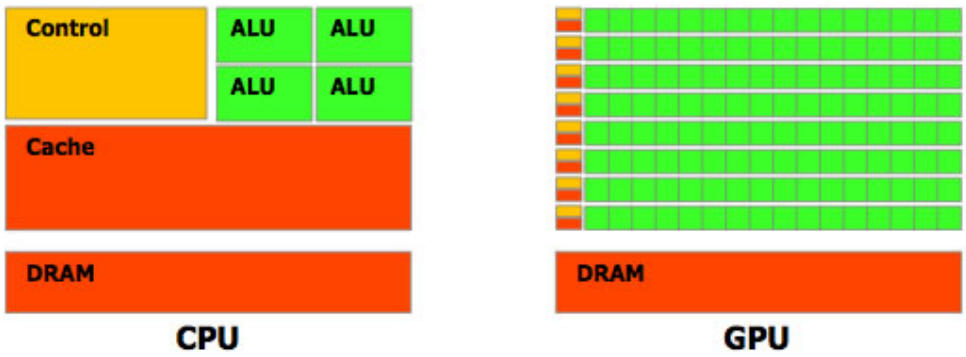

Figura 3.2: *Imagem comparando de forma simplifica a arquitetura dos processadores. Do lado esquerdo a arquitetura de uma Unidade de Processamento Central, muito espaço para o cache e a unidade de comtrole. Do lado direto uma unidade de processamento Gráfica, como uma grande região segmentada dedicada a computação aritimética e lógica. (Fonte: Wikimedia Commons [2])*

Na Figura 3.2 texto texto

era uma vez...

[2]Disponível em https://commons.wikimedia.org/wiki/File:Cpu-gpu.svg Creative Commons Attribution 3.0 Unported

3.4 Bibliotecas: OpenCL e CUDA

Open Computing Language (OpenCL)[3] é uma framework aberta para progra- mação genérica para vários processadores, dentre eles GPUs e CPUs. OpenCL da suporte para sistemas embarcados, sistemas pessoais, corporativos e até HPC. Ele consegue isso criando uma interface de baixo nível, ou seja, o mais próx- imo do hardware possível, e mantendo auto desempenho, com uma abstração portátil. O OpencL também é uma API para controle de aplicações paralelas em sistemas com processadores heterogêneos. O OpenCL consegue, numa mesma aplicação, reconhecer vários processadores diferentes dentro de um mesmo computador, e executar códigos distintos entre eles, coordenando os hardwares. Aqui, como no CUDA, a parte do código executado na CPU é chamada de Host e o hardware que executa os kernels de Devices . É importante lembrar que dado essa generalização do OpenCL, é possível que a CPU onde o código do host esteja executando seja usada para rodar um kernel, e essa CPU passa a ser um device ao mesmo tempo em que roda o host . Tanto o fato do OpenCL ser aberto quanto o fato dele não se restringir a um hardware especico fazem dele a linguagem mais usada para GPGPU fora de GPUs NVIDIA

Compute Unied Device Architecture (CUDA)[2] é uma arquitetura de progra- mação para GPUs criada pela NVIDIA. Ele adiciona suas diretrizes para as linguagens C, C++, FORTRAN e Java, permitindo que elas usem a GPU. Esse trabalho usa o CUDA junto com a linguagem C. A versão 1.0 do CUDA foi disponibilizada no inicio de 2007. Atualmente só existe um compilador para CUDA, o nvcc, e ele só da suporte para GPUs NVIDIA. Para uma função executar na GPU ela precisa ser invocada de um programa da CPU. Chamamos esse programa de Host e a GPU onde o kernel executará de Device . O CUDA implementa um conjunto virtual de instruções e memória, tornando os programas retroativos. O compilador primeiro compila o código em C para um intermediário, chamado de PTX, que depois será convertido em linguagem de máquina. Na conversão do PTX para linguagem de máquina o compilador verica quais instruções o device suporta e converte o código para usar as in- struções corretas. Para obter o maior desempenho possível, é importante saber para qual versão o código nal será compilado, pois na passagem do código de uma versão maior para uma menor não existe a garantia que o algoritmo seguira as mesmas instruções, o compilador pode mudar um conjunto de instruções para outro menos eciente, ou em alguns casos, algumas instruções não existem em versões mais antigas do hardware.

cuda mais facil de aprender

o que são e como funcionam porque escolhemos cuda? colocar código de cada uma

3.5 Aplicações e usos avançados

aprendiozado de máquina, deep learning, hadoop simulações climáticas realidade virtual renderização em tempo real de rostos humanos reais bitcoinbilhões de ciclos, no total, por segundo

Capítulo 4

Otimizacao

4.1 Arquitetura

mostrar gráfico de processamento do grmonty apontar o track super photon como candidato a ser produzido no kernel

4.2 Melhorias e Modificações

4.2.1 Somente uma dimensão

matrix pra vetor

4.2.2 OpenMP e Concorrência

desligar o openmp

4.2.3 math.h

unix math pra nvida math

4.2.4 divisão e trabalho e parelelização

calculo de diviasão de trabalho na GPU

4.2.5 Processar em Lotes

de “assim que possível” “para processamento em lotes”

Capítulo 5

Resultados

5.1 Métricas e Medição

the old

5.2 Comparações

demonstrar o aumento de 100X na velocidade

Capítulo 6

Futuro

6.1 Outras Linguagens de Programação

rust, nim, python

6.2 Single Precision

Usar float ao invés de double

6.3 Novos Disposítivos e Particulariedades dos frabicantes: AMD e NVIDIA

cálculo discreto, artiqueturas diferentes

6.4 Arcabouços: Tensorflow

tensorflow TPU TensorProcessingUnit

6.5 Application-specific integrated circuit chips (ASICs)

O que são? Onde vivem? O que comem?

Capítulo 7

Conclusões

Calculos são importantes e o avanço da ciência depende de artiteturas de alta performance, gpus tem se apresentado competentes na realização de tais tarefas, e sua popular adoção facilita um maior acesso computação astrofísica, aumentando assim a velocidade do progresso cientifico.

Referências Bibliográficas

Alfvén(1942) H. Alfvén. Existence of Electromagnetic-Hydrodynamic Waves. *nature*, 150: 405–406. doi: 10.1038/150405d0. Citado na pág. 3

Atari(1983) Atari. *Field Service Manual: Model 2600/2600A Domestic (M/N)*, 1983. Citado na pág. 10

Balraj(2016) Tarun Balraj. Nvidia geforce gtx 1070 goes on sale. https://www.techpowerup.com/223293/nvidia-geforce-gtx-1070-goes-on-sale, jun 2016. último acesso em 31/01/2018. Citado na pág. 8

Dolence *et al.***(2009)** Joshua C. Dolence, Charles F. Gammie, Monika Mościbrodzka e Po Kin Leung. grmonty: A monte carlo code for relativistic radiative transport. *The Astrophysical Journal Supplement*, 184:387–397. Citado na pág. 1, 3

Eijkhout *et al.***(2016)** Victor Eijkhout, Edmond Chow e V. Robert Geijn. *Introduction to High Performance Scientific Computing*. Saylor Academy. Citado na pág. 4

Fatahalian e Houston(2008) K. Fatahalian e M. Houston. Gpus: A closer look. Citado na pág. 9

GREEN *et al.***(2008)** S. GREEN, M. HOUSTON, D. LUEBKE, J. D. OWENS, J. C. PHILLIPS e J. E STONE. Gpu computing. *Proceedings of the IEEE*, 96. Citado na pág. 9

Hague(2013) James Hague. Why do dedicated game consoles exist? https://web.archive.org/web/20150504042057/http://prog21.dadgum.com/181.html, set 2013. último acesso em 4/5/2015. Citado na pág. 10

Intel(2016) Intel. Intel® core™ i7-6850k processor. https://ark.intel.com/products/94188/Intel-Core-i7-6850K-Processor-15M-Cache-up-to-3_80-GHz, jun 2016. último acesso em 31/01/2018. Citado na pág. 8

Jessen(2017) Steen Jessen. Big book of amiga hardware. http://www.bigbookofamigahardware.com/, 2017. último acesso em 31/01/2018. Citado na pág. 10

Kirk e mei Hwu(2016) David Kirk e Wen mei Hwu. *Programming Massively Parallel Processors*. Morgan Kaufmann, 3 edição. Citado na pág. 8, 9

McCall(1996) Scott McCall. N64's us launch. http://www.pennoaks.net/archive64/Miscellaneous_Articles/N64_US_Launch.htm, sep 1996. último acesso em 31/01/2018. Citado na pág. 10

Microsoft(2018) Microsoft. DirectX. https://msdn.microsoft.com/en-us/library/windows/desktop/hh309467, 2018. último acesso em 31/01/2018. Citado na pág. 9

Nintendoi of America Inc.(1998) Nintendoi of America Inc. *Nintendo64 Function Reference Manual*, 1998. Citado na pág. 10

The Khronos Group Inc.(2018a) The Khronos Group Inc. OpenGL. https://www.opengl.org/about/, 2018a. último acesso em 31/01/2018. Citado na pág. 9

The Khronos Group Inc.(2018b) The Khronos Group Inc. Vulkan. https://www.khronos.org/vulkan/, 2018b. último acesso em 31/01/2018. Citado na pág. 9