

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

Acelerando transporte radiativo ao redor de buracos negros com GPUs

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Rodrigo Nemmen da Silva (IAG/USP)

São Paulo
Dezembro de 2017

Resumo

Grmonty é um programa que calcula o espectro radioativo próximo a buracos negros. Para executá-lo é necessária muita paciência, uma vez que sua estrutura onera muito a CPU da máquina. Com o intuito de melhorar o desempenho do programa é empregada uma técnica de Computação de Alta Performance, na qual a carga de trabalho é dividida entre vários processadores, por meio das placas de gráficas, as quais possuem centenas de deles. Dividindo a carga de trabalho, rodando em paralelo, demonstramos uma aparente melhora de 3 vezes na velocidade de execução, fornecendo indícios de que esse método pode trazer grandes vantagens ao desempenho do programa.

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

Abstract

Grmonty is a software made to calculate the radioactive spectrum near black holes, but in order to execute it, patience is needed since it's structure is onerous to the machine's CPU. With the intention of improving the performance of the software a technique is applied from High Performance Computing, in which the workload is spread across multiple processors, through graphics cards, which posses hundreds of those. Splitting the execution, working in parallel, we show what seems to be 3 times improve in speed, providing evidence that the technique may bring gains in performance.

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

Sumário

Lista de Abreviaturas	vii
1 Introdução	1
2 Grmonty: Monte Carlo para Relatividade Geral	3
2.1 O que é o GRmonty	3
2.2 Como o programa funciona	4
3 GPGPU	7
3.1 Placas de vídeo	7
3.2 A História das GPU e GPGPU	10
3.3 CPU vs GPU	11
3.4 Bibliotecas: OpenCL e CUDA	14
3.5 Aplicações e usos avançados	15
4 Otimização	17
4.1 Arquitetura do Programa	17
4.2 Melhorias e Modificações	18
4.2.1 OpenMP e Concorrência	18
4.2.2 funções matemáticas	23
4.2.3 divisão e trabalho e paralelização	24
4.2.4 Processar em Lotes	25
5 Resultados	27
5.1 Métricas e Medição	27
5.2 Comparações	28
6 Conclusões	29
A grafo completo grmonty CUDA	31
Referências Bibliográficas	33

Listas de Abreviaturas

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

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 de 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 vêm se tornando 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 com o intuito 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**. Tornar o programa mais performático e lhe fornecer a capacidade de executar em computadores domésticos com dados e precisão relevantes, proporcionaria um avanço na pesquisa de buracos negros. Facilitando assim a execução de simulações e diminuindo o tempo de espera por resultados.

O **grmonty** também apresenta algumas outras características que o tornam atraente 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 parallelizar e distribuir a carga de trabalho pelos múltiplos núcleos de processamento destes dispositivos. 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 do desempenho, assim são apresentadas as otimizações executadas, chegando finalmente nos resultados alcançados e conclusões. Há ainda um capítulo de próximos passos indicando 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 é o GRmonty

Dolence et al definem o **grmonty** como “software destinado a calcular o espectro de plasmas quentes e opticamente finos levando em consideração a relatividade geral por completo, 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.

Buracos negros são regiões no espaço-tempo que possuem uma gravidade tão grande, que a velocidade de escape nestas áreas é maior que a velocidade da luz, assim nada é capaz de escapar deste local. Uma vez que algo adentra é impossível que saia Begelman (2010).

Existem diferentes categorias de buracos negros, podemos diferenciá-los de acordo com a sua massa. Existem os buracos negros estelares que são restos de estrelas, e os supermassivos que habitam o núcleo de galáxias. Quando uma estrela com mais de 25 vezes a massa do Sol, isto é, 25 massas solares torna-se uma supernova, o resto de seu núcleo pode se tornar um buraco negro estelar. Fortes evidências apontam que Cygnus X-1 com aproximadamente 14.8 massas solares é um buraco negro desse tipo Orosz *et al.* (2011).

No centro de nossa galáxia evidências apontam a existência de um buraco negro supermassivo Srg A*, com massa estimada em $4 \cdot 10^6$ massas solares Boehle *et al.* (2016), ou seja, sua massa é milhões de vezes maior que a do sol. Muito provavelmente sua formação não advém de uma supernova e sim da fusão de vários buracos negros Begelman (2010).

Buracos negros são geralmente rodeados por um gás composto de partículas carregadas, plasma. Este gás está em rotação ao redor do buraco formando um disco de acreção. Ele é submetido a gigantescas acelerações, sofrendo fricção, sendo comprimido e aquecido a temperaturas entre $10^4 k$ e $10^{12} k$. Devido a estas altas temperaturas, o plasma emite radiação eletromagnética em um amplo intervalo de comprimentos de ondas, do rádio a raios gama Lynden-Bell (1969).

Modelar e simular o espectro de radiação eletromagnética, gerado por estes discos de acreção tem sem mostrado um grande desafio. Este é o problema que o **grmonty** se propõem a resolver.

Quando é dito que o programa leva em consideração toda a relatividade geral, significa que todos os efeitos do espaço-tempo curvo, da relatividade geral de Einstein, são levados em conta na absorção, emissão e espalhamento dos fótons. Criando assim um espectro final mais detalhado e rico *Dolence et al. (2009)*

Utilizando o método de Monte Carlo na geração de fótons, e a partir de um dado modelo de plasma fornecendo a velocidade, densidade, intensidade do campo magnético e temperatura, o programa busca gerar um espectro da radiação emitida. Para tanto um número próximo a N - fornecido na entrada - de fótons é gerado e para cada um sua trajetória é traçada. Nesse percurso o fóton é espalhado podendo passar por diversas interações, até finalmente ser mensurado.

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

O objetivo do **grmonty** é quantificar o espectro eletromagnético de plasma ao redor de buracos negros, como função das condições do gás (velocidade, densidade, intensidade do campo magnético e temperatura) e das propriedades do buraco negro, massa, momento angular e carga elétrica *Dolence et al. (2009)*.

2.2 Como o programa funciona

No momento de criação e rastreio dos fótons o programa faz uso da biblioteca **OpenMP** para paralelizar o seu desenvolvimento. Graças a esta abordagem é viável o potencial aproveitamento 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. Isso só é possível por que, as instruções não dependem de cada fóton, elas são as mesmas para todos. Assim, podemos caracterizar o **grmonty** como tendo uma computação SIMD.

“Única Instrução Múltiplos Dados: Nesse tipo de computação podem existir múltiplos processadores, cada um operando sobre um 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 factível e simples de ser aplicada.

A parte do código que opera sobre os fótons apresenta características que facilitam a sua paralelização e portabilidade para uma arquitetura SIMD, como: não há escrita de variáveis compartilhadas entre os fótons, não há desvios de fluxo que tornem uma execução diferente das outras, não há necessidade de sincronização ou mutex (salvo no momento de incrementar

o número total de fótons).

Também existem alguns desafios à aplicação da paralelização. Os dados que representam as condições iniciais do plasma são compartilhados por todos os fótons. São variáveis globais acessadas por referência. Há condicionais que podem tornar alguma execução em particular mais longa, segurando o progresso das outras que estão na mesma unidade de processamento (caso das GPUs)

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, ou seja, não há criação de um conjunto de n fótons que depois teriam seu caminho rastreado, tornando a relação n para n . Isto agrupa funções muito diferentes, que acessam e dependem de variáveis e funções distintas, juntas. Acarretando uma maior dependência aos dados, que podem estar mais distantes indo contra o princípio da localidade, provocando uma possível perda de performance.

Esta relação é percebida ao se observar as linhas 106 a 137 do *grmonty.c*, 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 comando *make_super_photon* é executado, gerando um novo fóton *ph*, o *track_super_photon* é chamado, não havendo nenhum buffer ou lote para guardar um conjunto de fótons para um processamento posterior. Um fóton produzido é imediatamente consumido.

Tal processamento reduz a quantidade da memória principal reservada durante a execu-

ção do programa. Uma vez que o mesmo espaço alocado para um fóton, após seu processamento será utilizado pelo próximo, há um uso econômico de memória. O número de fótons em 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. Segundo são apontadas as diferenças arquiteturais entre uma CPU e uma GPU, vantagens e desvantagens. Na continuação são apresentadas às duas principais linguagens que permitem acesso à programação genérica na GPGPU, a OpenCL e a CUDA, indicando os pontos que pautaram a escolha de uma delas no desenvolvimento de um paralelismo massivo no grmonty. Por fim são apresentadas aplicações que se utilizam do estado da arte no hardware e no 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 desempenho a um baixo custo se comparado a CPUs, no que diz respeito a transformação de dados na memória em imagens e vídeos, prontos para serem apresentados em telas e monitores. São constituídas principalmente de:

- Um chip de memória, cuja 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 Multimidia Interface). São as saídas mais comuns às computações das placas.
- Um processador, uma GPGPU composta de algumas centenas a milhares de núcleos.
- Uma ou várias ventoinhas, para dissipar o calor produzido pelo processador.

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



Figura 3.1: Placa gráfica 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 a um preço de 379 dólares e possui 1920 núcleos de processamento à 1,506 GHz em sua GPU (Balraj, 2016). Nessa mesma época também foi comercializado o Intel® Core™ i7-6850K Processor CPU da Intel, vendida na época a partir de 617 dólares, possui 6 núcleos a 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 pela qual as GPUs passaram, onde 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 advém da sua função original de processar uma cadeia de dados gráficos. Esta 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 propriedades das fontes de luz. APIs gráficas como OpenGL (The Khronos Group Inc., 2018b), DirectX (Microsoft, 2018) ou Vulkan(The Khronos Group Inc., 2018c) representam este processo como um pipeline que executa uma cadeia de operações sobre um conjunto de vértices enviados pela CPU, sendo que cada vértice possui propriedades determinadas, tais como cor, posição e vetor normal (Fatahalian e Houston, 2008).

O vertex shader é uma parte integrante dessa cadeia e está no início do processamento. É um programa que executa um conjunto de operações sobre cada um dos vértices de entrada, projetando cada vértice, baseado em sua posição relativa à câmera virtual, em um anteparo 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 de forma paralela e/ou distribuída (GREEN *et al.*, 2008).

O próximo passo é o processo de rasterização, consistindo 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 contém todas as informações necessárias para gerar uma imagem final como profundidade, posição no frame buffer, etc. A partir da localização da câmera virtual, os fragmentos que são ocultos por outros fragmentos são descartados, só é necessário mostrar os objetos que podem ser vistos (GREEN *et al.*, 2008).

Por fim há o pixel shader que 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 esses programas podem ser aplicados em milhares de vértices e pixels independentemente, as GPUs evoluíram para ser um conjunto de multiprocessadores massivamente paralelos. Além disso, dependendo do balanceamento da carga de trabalho das aplicações, 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 parallelização, com mais núcleos, mais unidades lógico-aritméticas em detrimento de mais ciclos por segundo, maiores frequências de trabalho, priorizando uma única memória grande e não muito rápida ao invés de varias camadas de memória cache muito rápida.

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

3.2 A História das GPU e GPGPU

As GPUs, a princípio, não foram criadas para cálculos numéricos ou simulações de partículas e sim para a renderização de imagens para jogos digitais, o alto custo na produção e aquisição de hardware mais potente, principalmente a memória, na década de 70 impulsionou a criação de processadores mais dedicados e com funções mais específicas para a renderização 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 um conjunto de chips de vídeo para combinar os dados enquanto estes eram escaneados e direcionados ao monitor (Hague, 2013).

Em 1977 o Atari 2600 usou um *video shifter*, um tipo de circuito 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 circuito *blitter*, para maior velocidade na manipulação de memória, transformação de bitmaps, desenho de linhas e funções de preenchimento de áreas. Também vinha com um coprocessador, 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 década 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 distinção entre os dois principais processadores desses vídeo games eram claras, a nintendo já havia detalhado que seu sistema vinha com dois processadores: um de propósito mais geral o *MIPS R4200* e o *Reality Coprocessor* desenhado para cálculos em 3d de alta performance, pré-processamento de áudio 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 GPU para descrever seu *Reality Coprocessor*, foi a Nvidia em 1999 que popularizou o termo. Ele já existia desde década de 80, porém, somente após a campanha 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 (Nvidia, 1999).

Em 2002 foi introduzido a placa de vídeo ATI Radeon 9700 (também conhecida como R300)(Shimpi, 2002) nela tanto o pixel shader quanto o vertex shader poderiam implementar laços e longas contas aritméticas de ponto flutuante. A GPU foi se tornando tão flexível quanto as CPUs, porém em ordens de magnitude muito mais rápida em operações sobre listas, vetores.

Em 2007 a Nvidia lançou a CUDA, uma extensão da linguagem de programação C que poderia ser compilada e executada na GPU, oficializando o início da programação heterogênea nas placas de vídeo. CUDA foi o primeiro modelo de programação para GPU amplamente adotada, logo no ano seguinte surgiu o OpenCL que se tornou amplamente suportado. OpenCL é um padrão aberto definido pelo *Khronos Group* o qual permite o desenvolvimento de código tanto para a CPU quanto a GPU com ênfase na portabilidade (The Khronos Group Inc., 2018a). Programas nessa especificação podem rodar em dispositivos Intel, AMD, Nvidia e ARM.

Já em 2014 a Nvidia lança o GameWorks, um *middleware* que facilita e aumenta a performance da renderização de partículas como fumaça e fogo, faz simulações de fluidos e cálculos de *ray-tracing* ou rastreamento de raios (Nvidia, 2018), enquanto a AMD lançou o TressFX, biblioteca que permite a criação de pelos e cabelos que aparência muito próxima a realidade (AMD, 2018).

O gráfico 3.2 demonstra a performance de algumas das principais placas de vídeo da Nvidia em um período de 6 anos, de 2010 a 2016. O gráfico demonstra a tendência ainda muito forte de crescimento cada vez mais veloz do desempenho das GPUs. A demanda para processamento e performance gráfica não apresenta um limitante superior, isto é, não há um máximo de performance computacional que o mercado não possa absorver. além disso a cada nova versão a diferença da atual para a anterior é cada vez maior, não há previsão para o esfriamento na produção e desenvolvimento de GPUs mais e mais poderosas.

3.3 CPU vs GPU

Ambos os processadores tem a tarefa primordial de executar instruções manipulando a memória e performando cálculos aritméticos, porém suas necessidades particulares e os casos de uso que se encontraram redefiniram sua arquitetura, agora mesmo tendo a mesma origem suas peças de hardware são profundamente distintas, cada uma com suas vantagens e desvantagens sobre a outra.

Já em 2009, o poder computacional para operações em ponto flutuante de uma GPU era cerca de dez vezes maior ao de uma CPU, ver seção 3.1. Já a sua taxa de transferência era cerca de três vezes superior (Kirk e mei Hwu, 2016). Uma CPU é otimizada para desempenho de código sequencial, geralmente utilizam-se de uma lógica de controle mais sofisticada, fazendo com que a CPU seja projetada com mecanismos como previsão de desvios e exe-

²Disponível em <https://www.techspot.com/article/1191-nvidia-geforce-six-generations-tested/page6.html> visitada em 31/01/2018

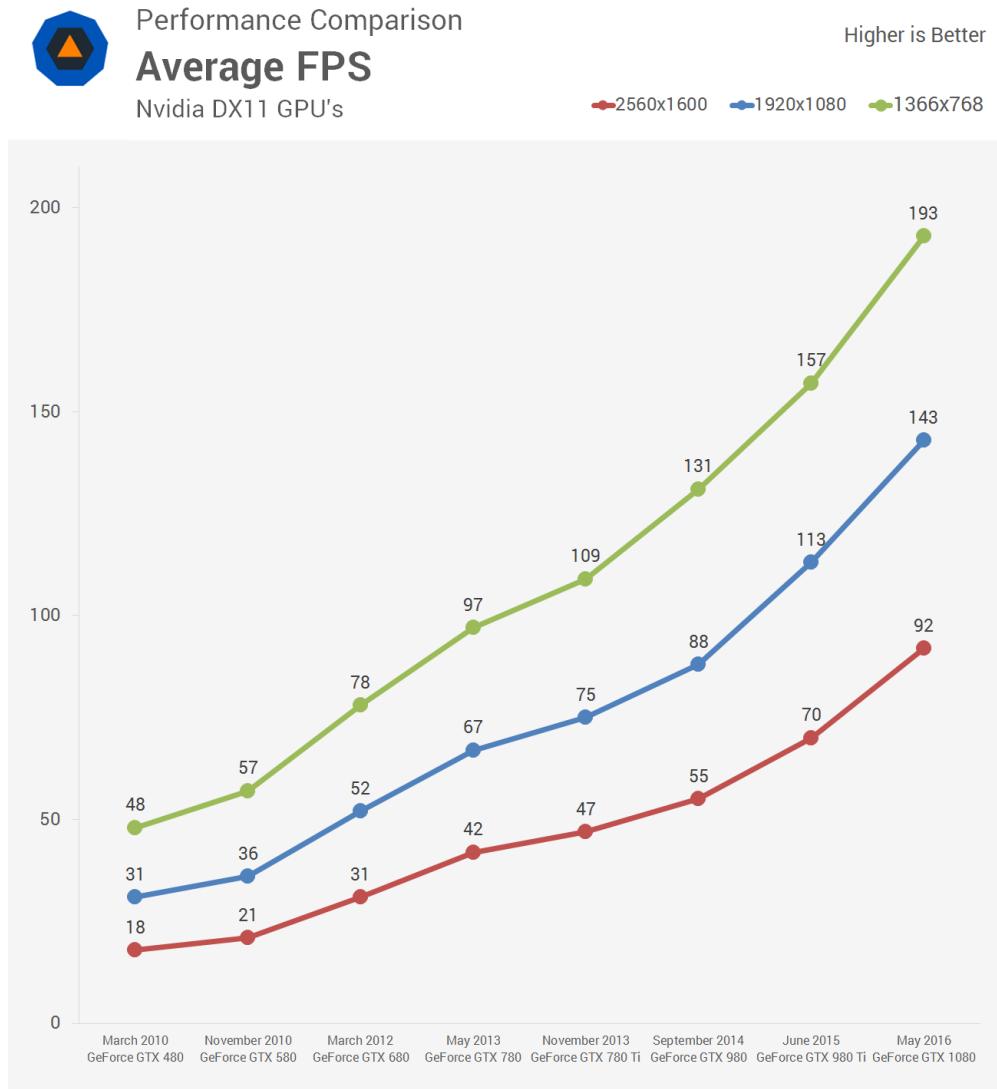


Figura 3.2: média de fps (frames por segundo) em alguns jogos modernos para as principais placas de vídeo da Nvidia que foram lançadas de 2010 a 2016 . (Fonte: TECHSPOT²)

cução fora de ordem (Out-of-order execution (PDF)). cs.washington.edu. 2006. Retrieved 17 January 2014. "don't wait for previous instructions to execute if this instruction does not depend on them"). 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, 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 níveis 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 e mei Hwu, 2016).

Já a GPU foi concebida para ser a solução de problemas no qual processamento dos dados pode ser realizado massivamente em paralelo, ou seja, o mesmo programa é executado simultaneamente em muitos conjuntos de dados com alta intensidade aritmética (NVIDIA,

2010a). Por causa disto, as GPUs dedicam a maior parte de seus recursos ao processamento aritmético ao invés de cache de dados ou controle de fluxo (Kirk e mei Hwu, 2016).

Na Figura 3.3, é exposta de maneira simplificada como que os transistores do hardware são distribuídos nos chips da CPU e GPU. Na imagem da esquerda há a representação de uma CPU, nela grande parte da área do chip é dedicada à memória cache e a lógica de controle. já na GPU, a imagem do lado direito, a maior parte dos recursos do chip é destinada ao processamento em paralelo de operações em ponto-flutuante. Assim, sua capacidade de processamento de dados é potencializada consideravelmente (NVIDIA, 2016).

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 e 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 processamento numérico, e não executam bem os programas sequenciais que as CPUs normalmente executam. Assim, deve-se utilizar a placa GPU como um coprocessador para auxiliar a CPU na execução dos programas, de forma que sejam atribuídas as partes sequenciais à CPU e as partes com processamento paralelo aritmético à GPU (Kirk e mei Hwu, 2016).



Figura 3.3: Imagem comparando de forma simplificada a arquitetura dos processadores. Do lado esquerdo a arquitetura de uma Unidade de Processamento Central, muito espaço para o cache e a unidade de controle. Do lado direito uma unidade de processamento Gráfica, como uma grande região segmentada dedicada a computação aritmética e lógica. (Fonte: Wikimedia Commons³)

³Disponível em <https://commons.wikimedia.org/wiki/File:Cpu-gpu.svg> Creative Commons Attribution 3.0 Unported

3.4 Bibliotecas: OpenCL e CUDA

Nos primórdios da computação genérica em GPUs duas formas principais surgiram para lidar com essa programação, a linguagem CUDA e a OpenCL.

Open Computing Language (OpenCL) é um arcabouço, um *framework*, aberto para programação genérica para vários processadores, não só GPGPUs como também CPUs e outros dispositivos focados em processamento. O OpenCL da suporte para sistemas embarcados, sistemas pessoais, corporativos e até de Computação de Alta Performance. Isso é alcançado graças a criação uma interface de baixo nível, ou seja, o mais próximo do hardware possível, mantendo um alto 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 e dispositivos diferentes dentro de um mesmo computador, ou cluster de computadores, e executar códigos distintos entre eles, coordenando os dispositivos. Desta forma, igual ao CUDA, a parte do código que é executada sempre na CPU, de forma sequencial é chamada de código *Host*. A parte que roda o *kernel*, o código que geralmente roda na GPGPU, preparado para rodar massivamente em paralelo é chamado de *device*. É importante lembrar que dado essa generalização do OpenCL, é possível que a CPU onde o código do host esteja executando seja também usada para rodar um kernel, e essa CPU passa a executar código device ao mesmo tempo em que roda código host. Tanto o fato do OpenCL ser aberto quanto o fato dele não se restringir a um hardware específico fazem dele a linguagem mais usada para GPGPU fora de GPUs NVIDIA ([Evan's data, 2011](#)).

Compute Unified Device Architecture (CUDA) pode ser descrita como uma linguagem de programação para GPGPUs criada pela NVIDIA em 2007 para seus processadores G80, fornecidos nas placas GeForce 8800 Ultra, GeForce 8800 GTX, GeForce 8800 GTS(G80). Ele adiciona suas diretrizes para as linguagens C, C++, FORTRAN e Java, permitindo que elas usem a GPGPU. Esse trabalho usa o CUDA junto com a linguagem C.

A versão 1.0 do CUDA foi disponibilizada em Fevereiro de 2007. Atualmente só existe um compilador para CUDA, o nvcc, e ele só da suporte para GPUs NVIDIA. Para uma função executar na GPGPU esta precisa ser invocada de um programa da CPU. Chamamos esse programa de Host e a GPGPU 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 verifica quais instruções o device suporta e converte o código para usar as instruções corretas. Para obter o maior desempenho possível, é importante saber para qual versão o código final será compilado, pois na passagem do código de uma versão maior para uma menor não existe a garantia que o código seguirá as mesmas instruções, o compilador pode mudar um conjunto de instruções para outro menos eficiente, ou em alguns casos, algumas instruções não existem em versões mais antigas.

Uma grande diferença entre essas duas linguagens é a quantidade e qualidade da documentação e recursos para o seu aprendizado. As duas apresentam basicamente a mesma estrutura e modelo de desenvolvimento, tornando razoavelmente simples para uma programadora ou um programador migrar de uma linguagem para a outra, porém a grande diferença nos recursos de aprendizado e documentação entre as duas, torna consideravelmente mais fácil e rápido aprender e desenvolver primeiramente com a linguagem CUDA.

Este é o principal motivador para que se tenha escolhido a linguagem CUDA para aumentar a performance do grmonty. O nosso grupo de estudos que é focado em aprimorar o grmonty tem a necessidade de ter um conjunto de pessoas com certo domínio sobre a linguagem que será usada no desenvolvimento, e graças a facilidade e agilidade do aprendizado da linguagem CUDA esta foi a escolhida que ser primeiramente empregada no aprimoramento de performance do grmonty.

3.5 Aplicações e usos avançados

No ano de 2017 não esta se observando uma retração no desenvolvimento das GPUs e sim uma expansão. Seja provocada pelo já conhecido mercado de jogos digitais, seja por novas fronteiras como mineração de cripto-moedas ou novas técnicas de aprendizado de máquina como *deep learning*. Abaixo são citadas algumas das aplicações e modelos de problemas que expandem o escopo e avançam o potencial que as GPUs podem alcançar.

O aprendizado profundo, *Deep Learning* é uma área de grande crescimento em inteligência artificial, ajudando os desenvolvedores a terem um controle mais fino e compreender melhor dados, como imagens, som e texto. Usando redes neurais, esses sistemas têm a capacidade de ver, aprender e reagir a situações complexas tão bem quanto ou melhor que os humanos treinados em tais ações. Isso está levando a uma forma totalmente diferente de pensar sobre coleta dados, tecnologias, produtos e serviços que podem ser oferecidos.

As soluções de aprendizado profundo contam quase exclusivamente com a computação acelerada pelas GPGPUs para treinar e acelerar aplicações, como identificação de imagens, caligrafia e vozes. As placas gráficas se destacam em pipelines paralelos e aceleram as redes neurais em até 10 a 20 vezes, reduzindo cada uma das muitas iterações de treinamento de dados de semanas para dias. As placas de vídeo têm acelerado o treinamento de redes neurais profundas em 50 vezes em apenas três anos — mais rápido do que a lei de Moore — com outras 10 vezes previstas para os próximos anos. A inovação nessa área está acontecendo a um ritmo acelerado, abrindo oportunidades em aplicações como robótica, medicina e carros auto-dirigíveis.

A realidade virtual (VR) é uma forma mais profunda de simular cenários mais realistas. Normalmente usando-se de capacetes ou óculos específicos, as vezes em combinação com objetos físicos reais no ambiente, imagens são geradas para cada um dos olhos e são atualizadas conforme o usuário move sua cabeça, proporcionando a sensação de estar corporalmente pre-

sente na cena gerada por computador. A pessoa portando esse equipamento pode "olhar ao redor" e interagir com itens e propriedades do mundo virtual.

O processamento necessário para gerar estas cenas de realidade virtual é atualmente muito alto, normalmente sendo exclusivos de dispositivos de primeira linha, uma vez que para que ocorra uma experiência razoavelmente imersiva se faz necessário haver um atraso desprezível na captação e apresentação das imagens, de acordo com o movimento do usuário, e as imagens devem ser atualizadas a uma frequência altíssima para que não ocorra enjoos da parte de quem usa tais aparelhos.

O processo de mineração de criptomoedas pode ser extremamente lucrativo se for feito de maneira muito rápida, uma vez que quanto mais blocos da *blockchain* você processar maior a sua chance de ganhar uma moeda no processo. Com o crescimento do interesse em bitcoins minerados - pessoas que buscam processar a blockchain, realizando transações das moedas - concluíram que as GPUs das placas gráficas usadas primordialmente para jogos poderiam ser usadas para resolver de forma muito mais rápidas os hash necessários para processar a blockchain.

A busca e o empenho dos mineradores em performar da melhor forma possível, empurrou o desenvolvimento das GPGPUs à criação de novas arquiteturas de placas gráficas e até a novos tipos de GPUs, quase ou nada focadas em gráficos. Esses novos dispositivos focados em mineração de criptomoedas se tornaram um novo foco de desenvolvimento e inovação na área de processadores massivamente paralelos de computação genérica, ou mais especificamente na resolução de hashes ou fatoração de números primos.

Capítulo 4

Otimização

O grmonty é um programa escrito em C que se utiliza fortemente da biblioteca OpenMP. Este trabalho é uma otimização de seu código, baseado principalmente em exportar o código para CUDA, mas para isso se faz necessário analisar a estrutura e funcionamento do programa, para assim encontrar pontos onde sua performance pode ser acrescida, usando principalmente a técnica de paralelização na GPGPU. Neste capítulo nos dedicamos a explicar a arquitetura na qual o grmonty opera e descrever pontos que poderiam ser otimizados e como poderiam, além de mostrar modificações necessárias para seu funcionamento na GPGPU.

4.1 Arquitetura do Programa

“Fazendo o design do grmonty nossa filosofia foi de maximizar a transparência física e minimizar o tamanho do código, ocasionalmente ao custo de redução de performance” ([Dolence et al., 2009](#)) (tradução nossa). Uma escolha técnica foi feita ao se desenvolver o grmonty, um enfoque maior na legibilidade de suas fórmulas físicas em detrimento da performance, uma escolha condizente com um código que se procura uma manutenção e estudo mais amigável, convidativa e simples.

Um exemplo dessa prática é ao computar um percurso por uma geodésica, tarefa computacionalmente dispendiosa. É amplamente reconhecido ([Xiao-Lin e Jian-Cheng, 2013](#)) que um esquema que depende da integrabilidade das geodésicas em um vácuo de Kerr é mais eficiente que integrar diretamente a equação da geodésica, porém essa segunda opção foi a escolhida pois é mais simples e mais curta ([Dolence et al., 2009](#)).

Todo o código do grmonty é escrito em C, tal escolha pode ter sido feita pois a linguagem é extremamente veloz e possui bibliotecas muito focadas em performance, além de permitir um controle mais fino da memória e das operações sendo executadas. Graças a essa característica a decisão de portar o código para CUDA é mais fácil pois CUDA pode ser visto como uma extensão de C.

O código do grmonty opera da forma mostrada na figura 4.1 seguindo o seguinte fluxo:

primeiro há um passo de carregamento dos dados onde o modelo com os dados de entrada é inicializado e variáveis auxiliares são inicializadas. Nesta etapa o processamento tem seu gargalo na leitura dos dados, a maior parte do tempo lê o modelo de entrada.

Após essa inicialização o programa entra em um laço principal (como pode ser visto no excerto de código na seção 2.2) onde duas funções são chamadas até que um número suficiente de fótons seja coletado. Como a geração de fótons provém do método de Monte Carlo, são produzidos em número aproximado, além disso muitos desses fótons criados podem não ser coletados, podendo ser perdidos e não contribuindo para o valor final.

Em um determinado momento é verificado que o número de fótons coletados atingiu um valor satisfatório então um condicional desvia o fluxo para a saída do laço principal e finalmente são produzidos os dados finais com o relatório da execução.

O cálculo computacional mais pesado que grmonty faz, e o fator de maior responsabilidade em seu tempo de execução é o rastreio de cada fóton. Este cálculo deve ser executado na ordem de milhões a centenas de milhões de vezes, onde cada vez é independente das demais. Essa arquitetura SIMD do programa pode ser explorada para um aumento de desempenho, aumentando a quantidade de execuções em paralelo que podem ocorrer. Maior paralelização maior velocidade.

Utilizando-se o OpenMP, a paralelização ocorre a nível de CPU, na qual o número de execuções simultâneas é da ordem de algumas unidades, a no máximo dezenas. Usando-se da tecnologia CUDA, a paralelização ocorre a nível de GPU, onde a quantidade de execuções simultâneas varia entre centenas a até milhares.

Dada a arquitetura do projeto, o foco principal para aumentar a performance é transformar a paralelização feita em OpenMP para uma outra feita em CUDA. Para isso se faz necessário que todo o código previamente distribuído pelas threads do OpenMP tornem-se Kernels, funções device, funções que executam exclusivamente na GPGPU. As principais funções do grmonty que são feitas sob o OpenMP são a *make_super_photon* e a *track_super_photon* que possuem as responsabilidades de gerar um fóton e rastrear a sua trajetória até o final, uma delas é também a função que demanda mais processamento. Para um N razoável capaz de gerar dados consistentes e interessantes 96,95% do tempo de execução é na função *track_super_photon* e sendo executada na ordem de centenas de milhares de vezes, tornando essa função a mais interessante para ser massivamente paralelizada.

4.2 Melhorias e Modificações

4.2.1 OpenMP e Concorrência

O grmonty foi originalmente desenvolvido com OpenMP para maximizar a paralelização, fazendo com que cada chamada a *make_super_photon* e *track_super_photon* fosse executada em paralelo e cada dupla de chamadas as duas funções rodaria em cada thread, isto só é possível porque cada fóton pode ser processado de maneira independente. Cada thread que

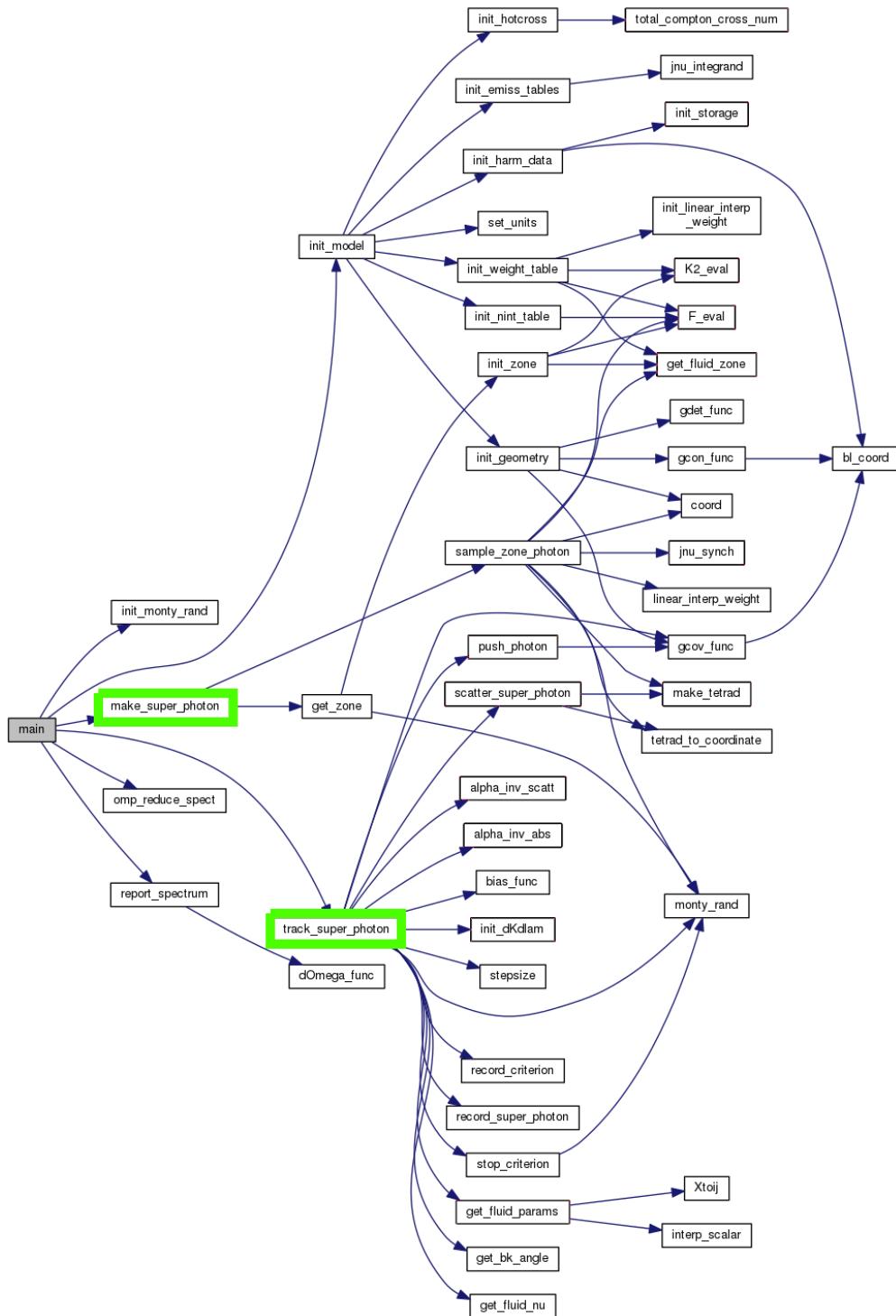


Figura 4.1: Grafo de chamadas de funções do grmonfy, com ênfase em verde para a track e a make super photon

cria e depois traça a trajetória das partículas é independente, porém ainda usam e consultam variáveis globais definidas no modelo criado na inicialização.

Entretanto o processamento de cada thread, de cada fóton, pode ou não resultar na coleta de um fóton (é possível que algum cálculo não converja a um resultado, assim o fóton associado é descartado), para verificar o ponto de parada do algoritmo, faz-se necessário que cada thread ao terminar, informe se obteve sucesso ou não, e se sim persistir os dados de seus

resultados para análise posterior. Assim existe um ponto no código onde todas as threads tem que parar e esperar, enquanto uma delas acrescenta o valor total de partículas coletadas e armazena os seus dados. Esta tarefa é relativamente simples em OpenMP, como o código do *record_super_photon* demonstra abaixo. Dois sinalizadores para o compilador fazem com que um trecho de código rode como em uma região crítica, uma região onde somente uma thread pode entrar por vez:

```

1 void record_super_photon(struct of_photon *ph)
2 {
3     /* inicializacao e verificacoes */
4     /* codigo codigo codigo... */
5
6 #pragma omp critical (MAXTAU)
7     {
8         /* max_tau_scatt eh variavel global externa*/
9         if (ph->tau_scatt > max_tau_scatt)
10             max_tau_scatt = ph->tau_scatt;
11     }
12
13     /* define em qual posicao salvar os dados */
14     /* codigo codigo codigo codigo... */
15
16 #pragma omp atomic
17     N_superph_recorded++; /* incrementa o numero fotons coletados */
18 #pragma omp atomic
19     N_scatt += ph->nscatt;
20
21     /* salva os dados */
22     /* codigo... */
23 }
```

É possível observar no código acima as diretivas do OpenMP, as quais são responsáveis por assegurar que o código definido sob elas irá ser executado por apenas uma thread por vez, nunca duas ao mesmo tempo. Nota-se também que são usadas duas diretivas diferentes, a `#pragma omp atomic` e a `#pragma omp critical (<TAG>)` (onde `<TAG>` pode ser um nome qualquer). Estas diretivas implementam funções de controle de região crítica, que não são triviais de serem reproduzidas em CUDA, uma vez que a arquitetura da GPU dificulta esse controle.

A implementação de regiões críticas em CUDA só pode ocorrer a nível de blocos de threads e não no nível threads individuais do mesmo bloco (mais sobre isso é explicado mais abaixo).

A biblioteca OpenMP possui algumas diretivas específicas que informam ao compilador

certas regras ao gerar código, as duas vistas anteriormente tem essa função. A `#pragma omp atomic` é mais simples, ela tem o objetivo de informar ao compilador que o trecho de código logo abaixo dela deve ser tratado como algo atômico, em outras palavras, que deve ser feito de uma única vez, não sendo permitido uma troca de contexto enquanto uma thread está sob essa diretiva. O outro pragma é um pouco mais complexo, é o `#pragma omp critical (<TAG>)`, é possível dizer que funciona como o anterior porém com alguns acréscimos. A primeira é a TAG, essa etiqueta é um nome qualquer escolhido pelo programador para identificar aquela região crítica de outras existentes no código. Enquanto o `omp atomic` para toda e qualquer thread de atuar, o `omp critical` somente vai impedir que outras threads no mesmo bloco identificado executem, isso lhe trás duas vantagens que a outra diretiva não tem: códigos mais complexos podem ser executados, algo que não é permitido no `omp atomic`, e o código lá executado não necessariamente vai bloquear toda a aplicação.

As diretivas do OpenMP funcionam por que são baseadas na POSIX threads api (POSIX), uma interface onde podem ser feitas chamadas ao sistema operacional e programas vão ser executados respeitando o padrão POSIX preestabelecido. Assim, já é conhecida e a muito tempo divulgada a interface com o CPU no que diz respeito a programação multithread, porém no universo das GPGPUs esses padrões não existem ou são muito novos.

Em CUDA a duas diretivas que o OpenMP faz podem ser implementadas, mas com uma dificuldade a mais. O `#pragma omp atomic` quando utilizado para a soma, como no código acima na linha 16 e 17 pode ser feito com a função CUDA `atomicAdd` onde é possível realizar uma soma em uma thread de forma atômica, sem que ocorra alguma condição de corrida entre as threads sendo possível atualizar o valor sem grandes problemas.

A implementação já se complica quando tentamos transcrever a parte do código da região crítica. O bloco em questão é o das linhas 6 a 11, nele existem duas ações de resolução complicada, o condicional e a atribuição.

A atribuição infelizmente não poderia ser implementada por um *atômicas* como nas linhas 16 e 17, pois mesmo não sendo uma soma poderíamos escrever como se fosse. O trecho:

```
1     max_tau_scatt = ph->tau_scatt;
```

poderia ser reescrito como:

```
1     max_tau_scatt += (- max_tau_scatt + ph->tau_scatt);
```

mas ainda assim não funcionaria pois o `atomicAdd` só realiza uma adição de forma atômica e a segunda forma precisa de duas adições e as duas tem quer ser feitas de uma vez. Isso é devido ao fato que ao se calcular o que deve ser adicionado a `max_tau_scatt` utiliza-se o próprio `max_tau_scatt` assim, é possível que o seu valor seja alterado por alguma outra thread entre o cálculo do o que deve ser adicionado e finalmente ser adicionado.

Este dado já suficiente para descartarmos o `atomicAdd`, porém implementar uma região crítica a esse nível na GPGPU não é uma tarefa trivial, e devemos implementar um condicional também. Existem formas de criar regiões criticas na GPGPUs, como por exemplo o

atomicCAS que realiza uma comparação e dependendo do resultado faz uma atribuição ou não, tudo isso de forma atômica.

Mas as threads em uma GPGPU não são independentes como as threads da CPU. As threads das GPGPUs e GPUs são agrupadas em grupos de 32, normalmente chamados de *warps*. Todas as threads em um mesmo warp executam as instruções em uma forma completamente *lock-step*, isto é, cada step que uma thread anda todas as outras também andam ao mesmo tempo. Se uma preposição de controle como um condicional ou um laço resulta em alguma ou algumas das 32 threads divergirem do resto, as threads remanescentes vão ficar em espera, dormindo, até as outras terminarem (Nvidia, 2012).

No caso de implementar a região crítica utilizando o *atomicCAS* para criar um mutex, deixaria algumas threads passarem enquanto outras ficariam esperando o mutex por um tempo indeterminado, uma vez que não há nada que reforce uma justiça fraca no escalonamento das threads da GPU, resultando num programa que nunca terminaria, entrando em deadlock.

Uma forma de resolver esse problema é permitindo que cada thread salve em um vetor compartilhado o valor de seu *ph->tau_scatt*, depois disso somente uma thread é encarregada de achar o maior valor e atribuir esse valor a *max_tau_scatt*. A implementação dessa resolução está abaixo:

```

1 void record_super_photon(struct of_photon *ph)
2 {
3     /* inicializacao e verificacoes */
4     /* codigo codigo codigo... */
5
6     my_max_tau[thread_uniq_id] = ph->tau_scatt
7
8     __syncthreads();
9
10    if( thread_uniq_id == 0 ) /* somente uma das threads executa esse
11        bloco */
12        for(size_t i = 0; i < max_threads; i++){
13            if(max_tau_scatt < my_max_tau[i])
14                max_tau_scatt = my_max_tau[i]
15
16            __syncthreads();
17
18            /* define em qual posicao salvar os dados */
19            /* codigo codigo codigo codigo... */
20
21            atomicAdd(&N_superph_recorded, 1);
22            atomicAdd(&N_scatt, ph->nscatt);
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
617
618
619
619
620
621
622
623
624
625
626
627
627
628
629
629
630
631
632
633
634
635
635
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
15
```

```
23  /* salva os dados */
24  /* código... */
25 }
```

A biblioteca OpenMP trouxe grande potência e simplicidade para o código do grmonty, mas ao se utilizar a programação em GPGPUs como CUDA, usar em conjunto essa biblioteca pode não ser somente mais complicado como prejudicial a aplicação.

Diferentemente da possibilidade de sempre ser possível criar um processo novo e lançá-lo a CPU, isso nem sempre é possível na GPU, na realidade o lançamento de diversos kernels a GPGPU sem que ela tenha terminado um antes de vir o outro, pode tornar todo o processamento mais lento ou matar a aplicação que estava sendo executada anteriormente. Por motivos como esses não é uma boa prática tentar lançar várias funções kernel através do OpenMP.

Como o OpenMP é amplamente usado no grmonty, e somente em algumas áreas específicas seria interessante usar a programação em GPGPU, o código OpenMP continua presente em algumas partes da aplicação. Os trechos de código mais computacionalmente pesados e com potencial para a paralelização são mais interessantes de serem portados para CUDA, deixando de utilizar o OpenMP.

Analizando o gráfico de processamento com indicação de performance, a figura 4.2 é evidente que o gargalo da aplicação é a função *track_super_photon*. Por que é que ela foi a escolhida para ser portada para a GPGPU, além de poder ser altamente paralelizável.

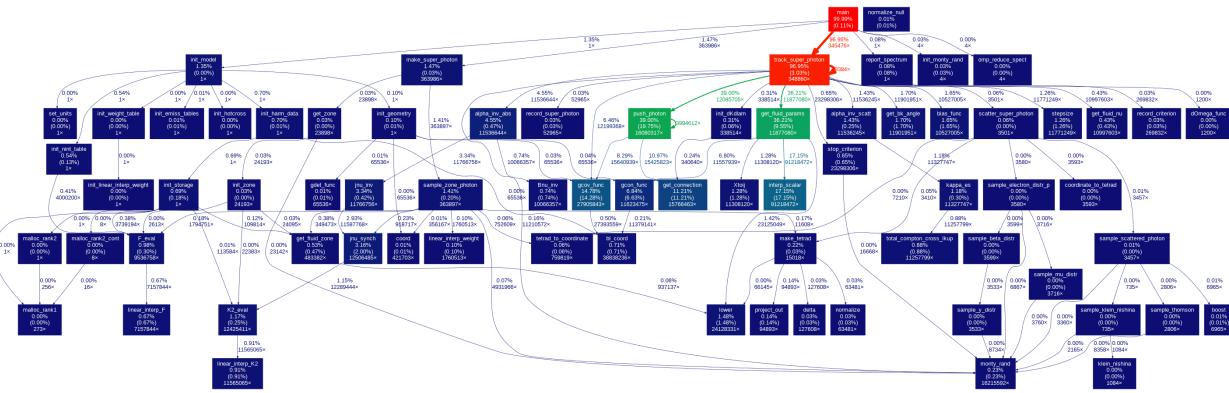


Figura 4.2: grafo de funcionamento do grmonty. Quanto mais vermelho mais tempo foi gasto. executados em um Ubuntu 17.10, x64, AMD Fx-6100 six-core processor, 8gb de ram. porcentagem tempo gasto até o retorno da função, porcentagem em parêntesis é o tempo gasto somente dentro da função, não contando outras chamadas.

4.2.2 funções matemáticas

Algumas funções matemáticas que o grmonty usa e que são implementadas pela gnu scientific library (gsl) não podem ser executadas na GPU uma vez que não existe suporte para essa biblioteca pela a Nvidia, porém existe uma biblioteca similar da própria nvidia,

a CUDAmath que apresenta uma grande coleção de funções matemáticas que podem servir como substitutas as da gsl.

O grmonty utiliza as funções:

- *int gsl_sf_bessel_Kn_e(int n, double x, gsl_sf_result * result)* método de calcula a função de Bessel cilíndrica modificada irregular de ordem n no ponto x
- *void gsl_ran_dir_3d(const gsl_rng * r, double * x, double * y, double * z)* Função que retorna um vetor normal de três dimensões aleatório com distribuição uniforme.
- *double gsl_rng_uniform(const gsl_rng * r)* Função que devolve um número aleatório de precisão dupla entre os valores de 0 e 1, excluindo ambos.

Destas a primeira e a última possuem uma função equivalente na biblioteca CUDAmath. Só existe uma ressalva sobre a primeira função, pois a gsl funciona para qualquer ordem N, enquanto a implementada pela cuda é somente para as ordens 0 e 1. Felizmente é possível construir as funções de ordem N a partir da de ordem 1.

A segunda função *void gsl_ran_dir_3d(const gsl_rng * r, double * x, double * y, double * z)* não contém uma equivalente em CUDAmath, por isso ela teve que ser implementada durante o desenvolvimento do código CUDA.

4.2.3 divisão e trabalho e paralelização

O grmonty paralleliza sua carga de trabalho via OpenMP, desta forma o programa tem total acesso a memória RAM e aos núcleos do processador. Todo o escalonamento de memória e processamento é administrado pelo sistema operacional. A partir do momento que o código passa a ser executado na GPGPU esse controle fino não tem como ser mais terceirizado ao sistema operacional, é de responsabilidade do programador administrar a memória e threads na GPGPU.

A GPGPU utilizada nos testes é uma Nvidia Geforce GTX 550TI, com 192 núcleos e 1024 Mb de memória. Para ser possível utilizar o máximo desse dispositivo é necessário mandar cargas de trabalho que utilizem todos os recursos na capacidade mais próxima a da máxima. Ao se utilizar, por exemplo uma carga de trabalho de 1100 Mbs em um primeiro passo de execução usaria 100% da memória porém na próxima usaria cerca de 10% da memória disponível, tornando 90% da memória ociosa, logo desperdiçando recursos. A mesma lógica pode ser aplicada aos núcleos da GPGPU, deve-se priorizar por dividir o número de threads que se deseja rodar por um múltiplo de 192, caso a diferença seja muito grande, ou seja a divisão possua um resto significativo, porém longe de 192, faz com que grande parte dos núcleos fiquem ociosos, desta forma desperdiçando recursos e tempo para realizar a alocação que não foi usada.

Essa peculiaridade da programação em GPUs torna cada programa responsável por se reconfigurar a cada computador novo ao qual se pretende executá-lo, porém é possível programar a alocação de recursos para que sempre seja a melhor possível.

Existe uma API na tecnologia CUDA que permite a busca por parâmetros dos dispositivos disponíveis para o processamento, assim todo o programa antes de lançar seus kernels pode buscar por informações do sistema em que está sendo executado e configura-se para otimizar o uso de recursos. por exemplo, para o uso otimizado do núcleos das GPGPUs é feita uma conta simples pra distribuir as threads em blocos na grid de execução das threads.

Quando se determina o número de threads necessárias para a execução ótima, deve-se dividir este número pela quantidade de blocos que se deseja executar e em quantas threads deve ter cada bloco.

Os núcleos das GPGPUs são divididas em uma *grid* e está grid é composta por blocos, cada um com uma quantidade de threads. Estes valores - de threads por bloco e número de blocos - são escolhidos no momento de lançamento do kernel e podem ter valores dinâmicos. Neste momento queremos dividir o número N de processos nessa arquitetura, onde o nosso objetivo é minimizar o número de threads ociosas. Cada bloco pode alocar somente uma potência de 2. Por exemplo, pode-se alocar 2, 8, 1024 ou 512 threads em um bloco mas não 300, 20 ou 1000, somente números que são potências de 2. A mesma regra vale para a quantidade de blocos, assim devemos encontrar o número de blocos $x = 2^k$ e a quantidade de threads por bloco $y = 2^p$ encontrando a solução mais aproximada possível para

$$N = 2^k * 2^p$$

4.2.4 Processar em Lotes

Uma mudança muito importante que deve ser feita para tornar possível a paralelização na GPU e torná-la mais eficiente é mudando o modo de operação do grmonty. Atualmente ele cria um fóton para o rastreio, é uma relação de um pra um, onde não há persistência do fóton, ou acumulo de fótons para a partir disto serem processados.

No modelo de GPGPU uma mudança para a maior performance seria trocar este modelo de processamento. Permitindo que ele possa ocorrer em forma de batchs (lotes de dados), isto é, um conjunto de fótons é produzido e armazenado para, somente no momento que se chegasse ao tamanho ótimo para ser executado na GPGPU, o kernel ser lançado. Para esta modificação se faz necessário armazenar inicialmente uma grande quantidade de fótons, executando o *make_super_photon* diversas vezes até que se consiga um tamanho ótimo para ocupar a GPU e que também faça o processamento ser eficiente na GPGPU.

Capítulo 5

Resultados

5.1 Métricas e Medição

Para medir o tempo de execução e processamento do programa utilizamos dois softwares. Principalmente o *gprof* para medir a performance das funções separadamente e o gnu *time* para medir o tempo total de execução dos programas.

Os testes foram executados em um Ubuntu 17.10, x64, AMD Fx-6100 six-core processor, 8gb de ram e uma placa de vídeo Nvida Geforce GTX 550TI, com 192 núcleos e 1024 Mb de memória. O número máximo de threads simultâneas na CPU é de 6 enquanto na GPGPU é 32 vezes esse valor, são 192 execuções paralelas no máximo.

O grafo 4.2 apresenta a ordem e tempo consumido em cada chamada de função no grmonty pré-otimização, nele é evidente o alto custo para se realizar o *track_super_photon*. Já o grafo 5.1 tenta mostrar o mesmo grafo só que para o código já portado para CUDA, porém a ferramenta utilizada a gprof, não foi capaz de medir a porcentagem de tempo que foi utilizada em funções que rodaram na GPGPU, mas mesmo assim é possível olhar a quantidade de vezes que a função foi chamada - no caso da *track_super_photon* ela foi chamada mais de 1.100.000 vezes para um N de 100 mil.

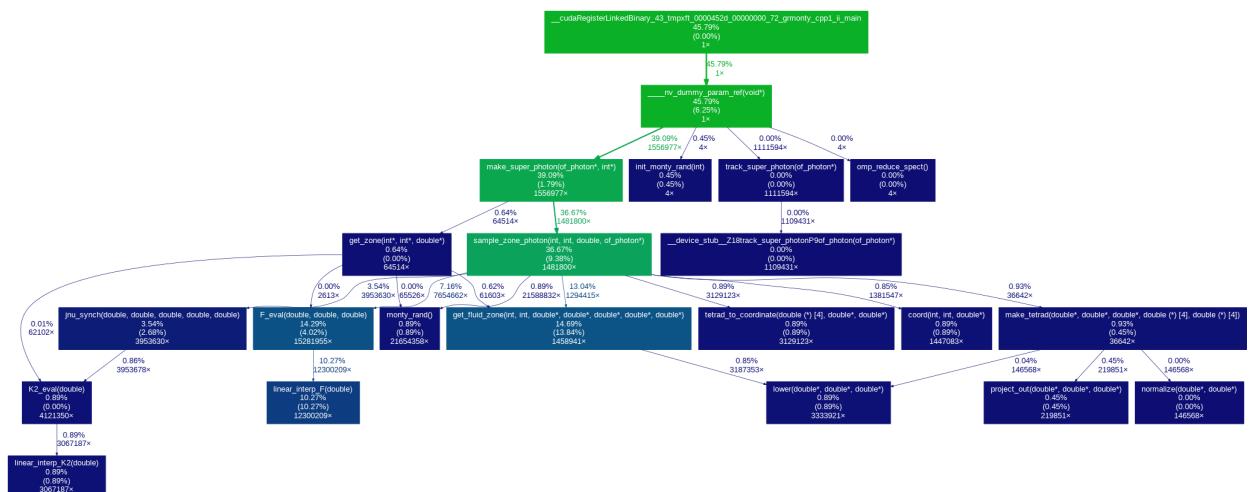


Figura 5.1: grafo de chamadas e porcentagem de tempo utilizado por função do grmonty em CUDA

O grafo em 5.1 é um recorte do grafo completo, já que ao se executar chamadas de código CUDA várias outras auxiliares, internas a arquitetura CUDA são invocadas, tal fato aumenta em muito o grafo e pode trazer um comprometimento a sua legibilidade, devido a este fato seu grafo esta suprimido aqui, mas está presente no apêndice.

5.2 Comparações

O gráfico abaixo 5.2 demonstra como o grmonty performa e como performa após ter parte de seu código portado para CUDA, conforme o número de fótons N cresce. Os pontos foram gerados executando-se as duas aplicações - a primeira versão completamente na CPU e a segunda com parte do processamento na GPGPU - Os valores de N vão de 0 a 100 mil pulando de 5mil em 5mil.

É fácil perceber que a versão em CUDA apresenta um tempo de execução bem menor que sua contrapartida na CPU. A diferença tende a 1/3, a versão em CUDA só toma 1 terço do tempo que a em GPU. É possível perceber também, olhando para o início do gráfico, quando N vale 0, que a versão em GPGPU possui um overhead maior, um maior tempo de inicialização que a outra versão.

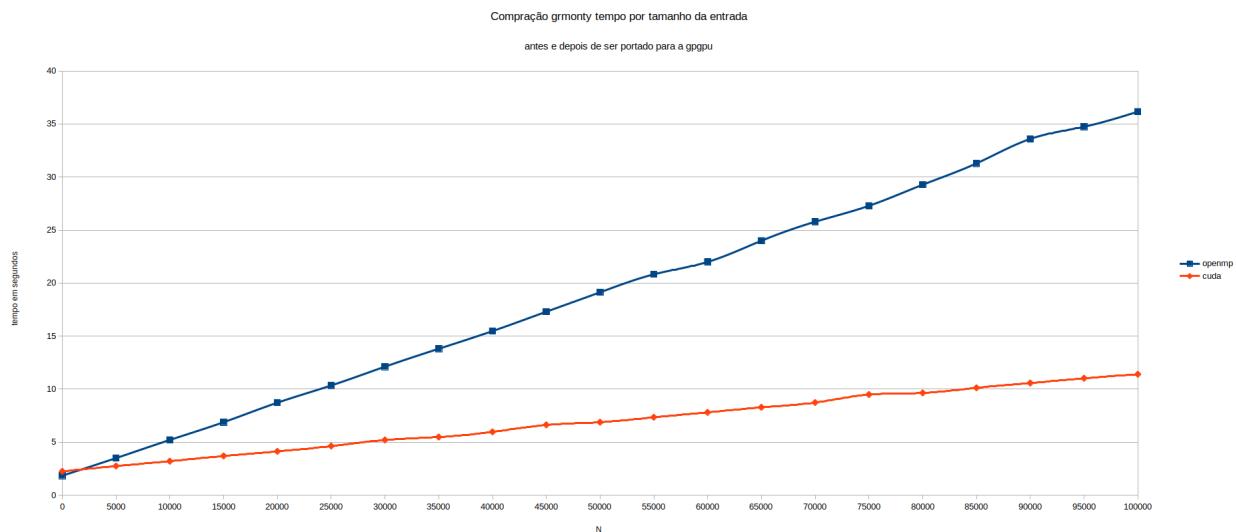


Figura 5.2: gráfico comparativo, antes e depois da otimização

Capítulo 6

Conclusões

Buscou-se nesse trabalho aplicar conhecimentos de arquiteturas de processadores, mais especificamente das GPGPUs, para aumentar a performance e diminuir o tempo de espera que o programa grmonty precisa para trazer resultados satisfatórios. Durante o seu desenvolvimento experimentos foram realizados e dados foram coletados resultando em um aparente *speed up* de 3 vezes, contribuindo para a ideia que a portabilidade de códigos para GPGPUS é uma forma válida e satisfatória para um aumento de performance para programas que podem ser altamente paralelizáveis.

A performance do grmonty está maior, ele responde mais rápido do que anteriormente. A utilização da GPGPU trouxe uma maior complexidade ao programa mas em contrapartida sua velocidade é muito maior. Ainda existe muito que pode ser feito para melhorar a performance, seja redimensionando o tamanho dos batches a serem processados seja uma serialização mais rigorosa nos kernels. Ainda existem recursões como no caso do *track_super_photon* que chama a si mesmo para recalcular uma trajetória caso passe por uma iteração que o espalhe novamente. Esses comportamentos afetam muito a performance pois todas as threads ficam esperando enquanto uma deles vai executar o *track_super_photon* todo novamente.

De acordo com (Kirk e mei Hwu, 2016) um ganho de 10 vezes é esperado em uma primeira transformação do código para a GPU, mas ganhos de até 100 vezes podem ser atingidos se um processo mais fino e delicado for aplicado, minimizando a alocação de memória na GPGPU e tornando a execução com a menor quantidade de desvios de fluxo possível, buscando tornar as threads mais uniformes possível.

Cálculos astrofísicos são importantes e o tempo que tomam para retornarem um resultado pode ser um limitante para a velocidade com a qual a ciência progride.

Sistemas gigantesco e de alto custo são construídos com o único propósito de computar dados para que se possam realizar experimentos, então é um enorme passo quando um pesquisador pode fazer tais cálculos muito complexos computacionalmente no mesmo computador onde joga seus jogos digitais no fim de semana.

O avanço da ciência depende de arquiteturas de alta performance, GPGPUs 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 científico.

Apêndice A

grafo completo grmonty CUDA

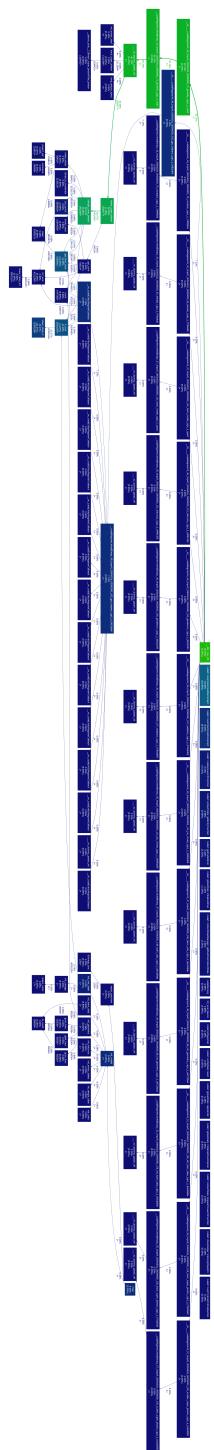


Figura A.1: grafo completo com todas as chamadas e com seus respectivos usos. código já em CUDA. Porcentagens iguais a zero não indicam que o program não execiou essas chamadas e sim que o g prof não foi capaz de medir o tempo gasto nelas

Referências Bibliográficas

- Alfvén(1942)** H. Alfvén. Existence of Electromagnetic-Hydrodynamic Waves. *nature*, 150: 405–406. doi: 10.1038/150405d0. Citado na pág. 3
- AMD(2018)** AMD. TressFX. <https://www.amd.com/pt/technologies/tressfx>, 2018. último acesso em 31/01/2018. Citado na pág. 11
- 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
- Begelman(2010)** Martin Begelman, Mitchell; Rees. *Gravity's Fatal Attraction, Black Holes in the Universe*. Cambridge University Press, second edição. ISBN 0521717930. Citado na pág. 3
- Boehle et al.(2016)** A. Boehle, A. M. Ghez, R. Schödel, L. Meyer, S. Yelda, S. Albers, G. D. Martinez, E. E. Becklin, T. Do, J. R. Lu, K. Matthews, M. R. Morris, B. Sitarski e G. Witzel. An Improved Distance and Mass Estimate for Sgr A* from a Multistar Orbit Analysis. 830:17. doi: 10.3847/0004-637X/830/1/17. Citado na pág. 3
- 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, 4, 17
- Eijkhout et al.(2016)** Victor Eijkhout, Edmond Chow e V. Robert Geijn. *Introduction to High Performance Scientific Computing*. Saylor Academy. citado na pág. 4
- Evan's data(2011)** Evan's data. Apac development study. Relatório técnico, Evan's data Corporation. Citado na pág. 14
- Fatahalian e Houston(2008)** K. Fatahalian e M. Houston. Gpus: A closer look. Citado na pág. 9, 13
- 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, 11, 12, 13, 29
- Lynden-Bell(1969)** D. Lynden-Bell. Galactic Nuclei as Collapsed Old Quasars. *nature*, 223:690–694. doi: 10.1038/223690a0. Citado na pág. 3
- 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
- Nintendo of America Inc.(1998)** Nintendo of America Inc. *Nintendo64 Function Reference Manual*, 1998. Citado na pág. 10
- NVIDIA(2016)** NVIDIA. Nvidia tesla p100: The most advanced datacenter accelerator ever built featuring pascal gp100, the world's fastest gpu. Relatório Técnico WP-08019-001, NVIDIA. URL <https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf>. Citado na pág. 13
- Nvidia(2012)** Nvidia. Cuda c best practices guide. Relatório técnico, NVIDIA. Citado na pág. 22
- Nvidia(2018)** Nvidia. GameWorks. <https://developer.nvidia.com/gameworks>, 2018. último acesso em 31/01/2018. Citado na pág. 11
- Nvidia(1999)** Nvidia. Nvidia launches the world's first graphics processing unit: Geforce 256. https://www.nvidia.com/object/IO_20020111_5424.html, 1999. último acesso em 31/01/2018. Citado na pág. 10
- Orosz et al.(2011)** J. A. Orosz, J. E. McClintock, J. P. Aufdenberg, R. A. Remillard, M. J. Reid, R. Narayan e L. Gou. The Mass of the Black Hole in Cygnus X-1. 742:84. doi: 10.1088/0004-637X/742/2/84. Citado na pág. 3
- Shimpi(2002)** Anand Lal Shimpi. Ati's radeon 9700 (r300) – crowning the new king. <https://www.anandtech.com/show/947>, jul 2002. último acesso em 31/01/2018. Citado na pág. 11
- The Khronos Group Inc.(2018a)** The Khronos Group Inc. OpenCL. <https://www.khronos.org/opencl/>, 2018a. último acesso em 31/01/2018. Citado na pág. 11
- The Khronos Group Inc.(2018b)** The Khronos Group Inc. OpenGL. <https://www.opengl.org/about/>, 2018b. último acesso em 31/01/2018. Citado na pág. 9
- The Khronos Group Inc.(2018c)** The Khronos Group Inc. Vulkan. <https://www.khronos.org/vulkan/>, 2018c. último acesso em 31/01/2018. Citado na pág. 9
- Xiao-Lin e Jian-Cheng(2013)** Yang Xiao-Lin e Wang Jian-Cheng. ynogkm: A new public code for calculating time-like geodesics in the kerr-newmann spacetime. Citado na pág. 17