UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

DESENVOLVIMENTO DO PSIRT PARALELO EM CUDA

ALUNO: LEONARDO MENDES PRIMO BRITO

ORIENTADOR: DR. SÍLVIO DE BARROS MELO

RECIFE, AGOSTO DE 2013

LEONARDO MENDES PRIMO BRITO

DESENVOLVIMENTO DO PSIRT PARALELO EM CUDA

Trabalho apresentado ao programa de graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Prof. Orientador: Sílvio de Barros Melo

Recife, agosto de 2013

ASSINATURAS

O presente estudo, intitulado *Desenvolvimento do PSIRT Paralelo em CUDA*, é fruto do trabalho do estudante Leonardo Mendes Primo Brito, sob orientação do prof. Sílvio de Barros Melo. As pessoas abaixo listadas reconhecem este Trabalho de Graduação e seus resultados.

Leonardo Mendes Primo Brito

Sílvio de Barros Melo

AGRADECIMENTOS

Agradeço a Deus pela oportunidade de apresentar este trabalho,

A meus pais, pelo exemplo acadêmico, pela paciência e apoio ao longo da graduação,

Ao meu orientador, prof. Sílvio Melo,

Ao amigo Ícaro Moreira, por gentilmente se dispor a explicar-me o PSIRT.

A beleza salvará o mundo

Fiódor Dostoiévski

RESUMO

Este trabalho de graduação apresenta um novo algoritmo de reconstrução tomográfica baseado no PSIRT (*Particle system iterative reconstruction technique*) e utilizando técnicas de GPGPU (*General-purpose computing on graphics processing units*) através da tecnologia CUDA. O objetivo deste trabalho é obter um ganho de desempenho considerável com o novo algoritmo CUDA.

ABSTRACT

This work presents a new tomographic image reconstruction algorithm based on PSIRT (*Particle system iterative reconstruction technique*) using GPGPU techniques, namely the CUDA architecture. This work's goal is to achieve a sensible speedup with the new CUDA algorithm.

Keywords: PSIRT, tomography, reconstruction, CUDA, GPGPU, parallel processing.

LISTA DE ILUSTRAÇÕES

Figura 1 - Diferentes gerações de tomógrafos [4] 19
Figura 2 – Esquema básico de um tomógrafo 20
Figura 3 - Tomógrafo 3x7 21
Figura 4 - Fantoma a ser reconstruído 23
Figura 5 - Reconstrução sem otimização 23
Figura 6 – Fantoma a ser reconstruído 24
Figura 7 - Reconstrução sem otimização 24
Figura 8 - Reconstrução com otimização 24
Figura 9 - Etapa de otimização do PSIRT 25
Figura 10 - Fantoma de um anel 27
Figura 11 - Execução do PSIRT 27
Figura 12 - Imagem reconstruída 27
Figura 13 – Hierarquia de threads CUDA [10, p. 9] 30
Figura 14 – Hierarquia de memória CUDA. [7, p. 11] 31
Figura 15 - Grids, blocos e threads executando em paralelo 34
Figura 16 – Fantomas utilizados nos testes de desempenho 39
Figura 17 - Tempo de execução em milisegundos para os 3 fantomas 39
Figura 18- Tempo de execução para os 3 fantomas (escala log) 40
Figura 19 – Speedups mensurados e projetados 41
Figura 20 - Melhor caso, pior caso e média aritmética 41
Figura 21 - Melhores e piores casos de CPU e GPU (escala logarítmica) 42
Figura 22 - Tempos de execução do fantoma 1 42
Figura 23 - Tempos de execução do fantoma 3 42

LISTA DE CÓDIGOS

Código 1 - Pseudocódigo do PSIRT sequencial 22

Código 2 - Pseudocódigo de otimização do PSIRT 24

Código 3 - Loop principal do PPSIRT, executado no host 33

Código 4- Trecho de otimização do kernel ppsirt 37

LISTA DE TABELAS

Tabela 1 - Tipos principais de tomografia 13

Tabela 2 - Saída do gprof para PSIRT em C 26

Tabela 3 – Hierarquia de memória CUDA 30

Tabela 4 - Profile do ppsirt 38

SUMÁRIO

1. Introdução 13

1.1. História da tomografia 13

1.2. Computação de alto desempenho 14

1.3. Computação em placas gráficas 15

1.4. Motivação 16

1.5. Objetivos 17

1.6. Estrutura 17

2. Fundamentos Teóricos 18

2.1. Tomógrafo – visão geral 18

2.2. Tomógrafo – base física e matemática 21

3. PSIRT 22

3.1. Otimização 24

3.2. PSIRT em C 25

3.2.1. Profiling 26

3.2.2. Discretização em pixels 27

4. CUDA 28

4.1. Modelo de programação 29

5. Parallel PSIRT (PPSIRT) 32

5.1. Detalhamento dos kernels 33

5.1.1. ppsirt 33

5.1.2. ppsirt_chkstable 34

5.1.3. ppsirt_update_traj e ppsirt_zero_traj 34
5.2. Hierarquia do PPSIRT 34
5.3. Otimização no PPSIRT 35
5.4. Sincronização do PPSIRT 35
6. Resultados 38
7. Conclusão 43
7.1. Limitações 43
7.2. Estudos futuros 43
Referências 45

1. INTRODUÇÃO

Neste capítulo faremos uma breve exposição das bases conceituais para o entendimento do trabalho. Especificamente, começaremos por uma rápida revisão da história da tomografia computadorizada e uma introdução à computação de alto desempenho. Posteriomente, segue a motivação para a realização do PSIRT paralelo, os objetivos do trabalho e a apresentação da estrutura desta dissertação.

1.1. HISTÓRIA DA TOMOGRAFIA

Tomografia, junção das palavras gregas *tomos* (“seção”) e *graphein* (“escrever”) é o nome dado à técnica de produzir uma imagem de um objeto a partir de seu seccionamento, utilizando para tal algum tipo de onda eletromagnética. O método foi teorizado nos anos 1930 pelo radiologista italiano Alessandro Vallebona. Atualmente a tomografia é utilizada numa plêiade de aplicações: biologia, medicina, arqueologia, geofísica, ciência dos materiais, astrofísica, entre outros. Alguns dos principais tipos de tomografia:

Onda	**Tipo de tomógrafo**
Raios X	Tomografia Computadorizada (CT – Computed Tomography)
Raios gama	Tomografia Computadorizada por emissão de fóton único (SPECT – Single photon emission computed tomography)
Ondas de rádio	Ressonância magnética (MRI – Magnetic resonance imaging)
Aniquilação pósitron-elétron	Tomografia por emissão de pósitrons (PET – Positron emission tomography)

Tabela 1 - Tipos principais de tomografia

Um dos precursores mais antigos, simples e populares da tomografia é a radiografia por raios X. A técnica, inventada em 1895 pelo alemão Wilhelm Röntgen, foi rapidamente absorvida pela academia e sociedade: menos de um mês após concluir seu paper no assunto, a primeira radiografia por raios X para uso médico foi realizada. Desde então, bilhões de radiografias foram feitas pelo mundo. A técnica abriu espaço para a tomografia por raios X, que consiste basicamente em várias radiografias tiradas do mesmo objeto mas de ângulos e posições diferentes. Com a criação e rápida evolução dos computadores, surgiu a *tomografia computadorizada*. Utilizando-se do poder computacional para operar o tomógrafo e processar os resultados, a tomografia computadorizada abriu um leque imenso de possibilidades de aplicações para a tomografia.

1.2. COMPUTAÇÃO DE ALTO DESEMPENHO

A computação de alta performance é uma área da ciência da computação envolvendo as técnicas de hardware e software necessárias para obter a máxima performance possível num algoritmo. Atualmente, tenta-se alcançar alto desempenho através da execução paralela do algoritmo.

Na esfera do hardware, um protagonista histórico desta área foi o *supercomputador*. O termo remonta aos anos 1960, quando o engenheiro americano Seymour Cray desenvolveu uma série de computadores para o *Control Data Corporation*, empresa americana de *mainframes*. O uso de paralelismo no processamento era uma das principais inovações técnicas usadas para obter maior desempenho na execução de software.

Na esfera do software, há inúmeras bibliotecas, compiladores e padrões utilizados para auxiliar o programador na tarefa muitas vezes não-trivial de paralelizar um algoritmo. Entre os padrões, há dois que podem ser tomados como referência afim de exemplificar diversas técnicas da área: *OpenMP* e *MPI.*

O MPI (*Message Passing Interface*) é uma especificação para processamento paralelo em sistemas de memória não-compartilhada através de passagem de mensagens entre cada unidade processadora independente, com implementações em C e Fortran. Fica a cargo do programador a definição de onde, quando e para quem cada CPU enviará cada mensagem de modo que o

algoritmo funcione corretamente em paralelo. A necessidade do tráfego de mensagens entre as unidades vem do fato de não haver no sistema uma memória compartilhada – encaixam-se neste critério, por exemplo, os supercomputadores tradicionais, onde cada célula é uma unidade independente, com memória própria (e.g. um conjunto de centenas de computadores em rede, cada um com uma única CPU e memória individual).

O OpenMP (*Open MultiProcessing*) é uma biblioteca para C, C++ e Fortran que implementa o paralelismo em sistemas de memória compartilhada através da criação de threads, distribuição de trabalho entre as CPUs e sincronização. Em C, isto é feito através de extensões da linguagem (*#pragma*). Com a recente popularização de CPUs multi-núcleo, criou-se demanda para paralelização de sistemas de memória compartilhada.

Ambas as técnicas são, portanto, complementares, e ilustram bem os principais desafios da área de computação paralela. Entre os anos 1960 e 2000, supercomputadores eram por via de regra redes de processadores cuja memória não é compartilhada. O surgimento e recrudescimento de CPUs de vários *cores* complicou este sistema: supercomputadores atuais são ao mesmo tempo de memória compartilhada e não-compartilhada. Diante deste contexto, conclui-se que não é mais suficiente o uso de apenas uma técnica.

1.3. COMPUTAÇÃO EM PLACAS GRÁFICAS

Placas gráficas (*GPU – Graphics processing unit*) são circuitos eletrônicos com objetivo específico de renderizar cenas gráficas rapidamente. Para cumprir este papel, a GPU deve realizar certas operações matemáticas com grande velocidade. Considerando a maneira como os gráficos de computador são representados, a maneira mais eficiente de realizar as transformações e operações matemáticas necessárias com grande velocidade é através do paralelismo massivo – o uso de centenas ou milhares de unidades lógicas-aritméticas (ALUs).

O grande crescimento da indústria de jogos, que em 2012 movimentou mais de US$ 20 bilhões [1], trouxe consigo jogos com gráficos cada vez mais detalhados. Com jogos cada vez mais exigentes e um mercado consumidor cada vez maior, nas últimas décadas a indústria de placas gráficas viu a demanda por GPUs de alto desempenho crescer constantemente, estimulando a inovação e desenvolvimento de técnicas cada vez mais eficientes. Por outro lado,

não houve fenômeno semelhante para as CPUs: a demanda de poder de processamento cresceu lentamente quando comparado às GPUs, pois as aplicações típicas do mercado consumidor não tornaram-se essencialmente mais complexas, como houve no caso dos jogos. Ou seja: enquanto há no mercado milhões de clientes ávidos por GPUs de altíssimo desempenho, os clientes a procura por CPUs de alto desempenho restringem-se basicamente a grandes empresas e instituições de ensino e pesquisa.

Dito isto, o imenso e crescente potencial de processamento das GPUs atrai, desde meados da década de 2000, cada vez mais interessados em usá-lo outras aplicações fora jogos. Surge uma nova área: computação de propósito geral em unidades de processamento gráfico, ou *GPGPU – General purpose computing on graphics processing units*. Percebeu-se que o formato dos problemas gráficos usualmente resolvidos por placas gráficas, que envolve matrizes e transformações lineares, são muito semelhantes aos formatos de vários outros problemas de diversas áreas. Além disso, houve um esforço por parte da indústria e da comunidade científica para o desenvolvimento de padrões, compiladores e bibliotecas que aproximem o máximo possível a programação em GPUs da programação tradicional para CPUs, resultando em produtos como *CUDA* e *OpenCL*.

Apesar de muito nova, a técnica já conta com aplicações em diversas áreas: bioinformática, dinâmica molecular, física, inteligência artificial, tomografia computadorizada, visão computacional, finanças, criptografia, computação científica, entre outras. Vários destes problemas alcançam *speedups* de mais de 1000x em relação ao algoritmo sequencial executado numa CPU [2].

1.4. MOTIVAÇÃO

Em aplicações reais, o PSIRT pode se mostrar demasiadamente lento para ser viável. Uma implementação paralela, portanto, teoricamente será de grande valia. A escolha pelo uso específico de GPGPU (em detrimento, por exemplo, de paralelismo em clusters ou em CPUs de multi-núcleos) dá-se principalmente por dois fatores: (i) baixo custo, (ii) alto potencial de ganho de performance. Enquanto a paralelização em CPUs normalmente trás ganhos lineares de acordo

com a quantidade de processadores, implicando em grandes gastos, a paralelização em CUDA pode, teoricamente, trazer ganhos de ordens de grandeza em performance a um custo fixo.

1.5. OBJETIVOS

O objetivo deste trabalho é projetar e executar uma implementação do PSIRT em CUDA-C que traga ganhos de performance significativos em relação à implementação em C executada numa CPU. Para isso, estuda-se o algoritmo sequencial, seus potenciais gargalos e as oportunidades de otimização e paralelização.

1.6. ESTRUTURA

O trabalho está dividido nos capítulos seguintes:

Capítulo 2: Fundamentos Teóricos. Nele serão apresentados os conceitos, técnicas e tecnologias relevantes ao trabalho.

Capítulo 3: PSIRT. É apresentado o algoritmo PSIRT, desenvolvido por [3]. É apresntada uma implementação original em C, que servirá de base comparativa futura, e estuda-se em profundidade quais as principais limitações e gargalos do algoritmo sequencial.

Capítulo 4: CUDA. Explica-se os principais conceitos necessários ao entendimento básico da tecnologia CUDA.

Capítulo 5: Parallel PSIRT. Apresenta-se o algoritmo PPSIRT. Discutem-se as estratégias de paralelização, os recursos CUDA utilizados e os desafios encontrados.

Capítulo 6: Resultados. Analisa-se os ganhos de desempenho obtidos com o novo algoritmo apresentado no capítulo anterior, em relação à implementação em C.

Capítulo 7: Conclusões.

Capítulo 8: Estudos futuros.

2. FUNDAMENTOS TEÓRICOS

Neste capítulo apresentaremos a configuração básica de um tomógrafo, os modelos físico e matemático do processo de reconstrução, alguns algoritmos de reconstrução relevantes e fundamentos de CUDA.

2.1. TOMÓGRAFO – VISÃO GERAL

O funcionamento básico de um tomógrafo pode-se resumir ao seguinte: uma fonte de radiação, denominada daqui em diante *emissor*, emite feixes de ondas direcionadas ao objeto a ser estudado, denominado *amostra*. Um instrumento de medição posicionado no extremo oposto ao emissor, denominado *receptor*, recebe as ondas *atenuadas*, i.e. com intensidade menor ou igual à intensidade emitida pelo receptor. Cada feixe de onda é chamado de *trajetória*, e um conjunto de trajetórias é denominado *projeção*. A *configuração do tomógrafo* é o número de projeções e a quantidade de trajetórias de cada projeção; um tomógrafo de configuração *n x m* tem n projeções equiangulares com m trajetórias cada.

A diferença entre a radiação emitida pelo emissor e a radiação recebida pelo receptor é chamada *atenuação*. De modo geral, quanto maior a densidade e volume de matéria entre o emissor e receptor, maior será a perda de energia do feixe, ou seja, maior será a atenuação. Ao conjunto de atenuações de todas as trajetórias dá-se o nome *sinograma*.

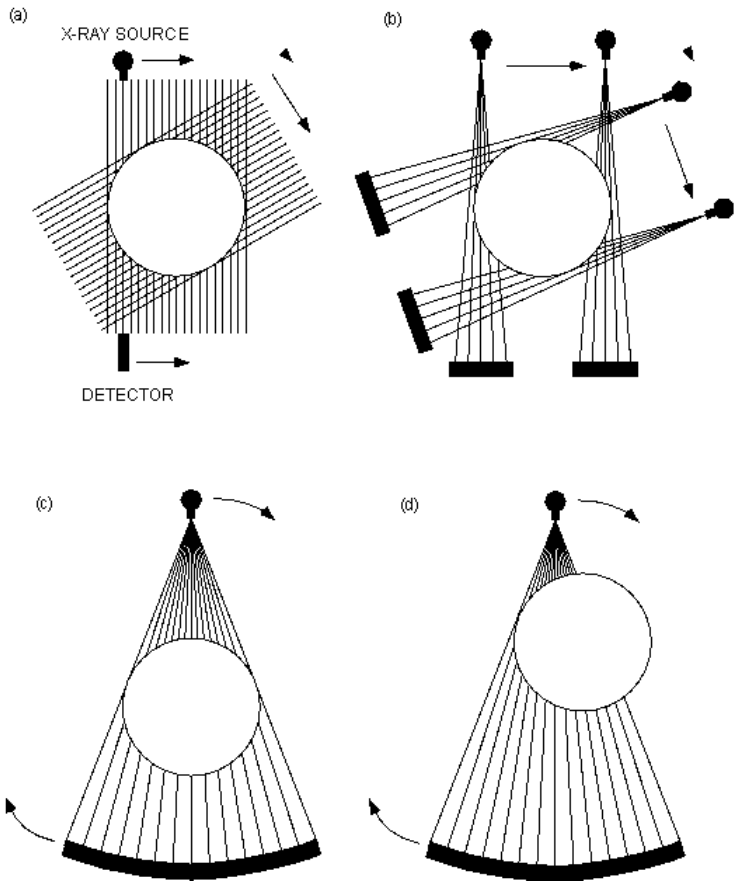

Figura 1 - Diferentes gerações de tomógrafos [4]

Há várias maneiras de se movimentar o conjunto emissor-receptor de modo a obter um sinograma. Na literatura, é comum dividir os tomógrafos em quatro gerações, de acordo com os tipos de movimentos efetuados. A figura 1 ilustra cada uma das gerações. São elas:

- Tomógrafos de primeira geração (Figura 2-a): cada projeção consiste em trajetórias paralelas. O par emissor-receptor move-se de maneira circular, ao redor da amostra.
- Tomógrafos de segunda geração (Figura 2-b): fanbeam de feixe estreito. Cada projeção consiste em trajetórias separadas por um mesmo pequeno ângulo.
- Tomógrafos de Terceira e quarta geração (Figura 2-c, d): fanbeam com feixe largo e rotação.

O PSIRT foi simulado tanto com tomógrafos de feixes paralelos quanto de feixes fanbeam [5]. Como o foco deste trabalho não é especificamente o tomógrafo, mas sim uma abordagem paralela do PSIRT, de agora em diante consideraremos apenas o tomógrafo de feixes paralelos, de entendimento mais simples.

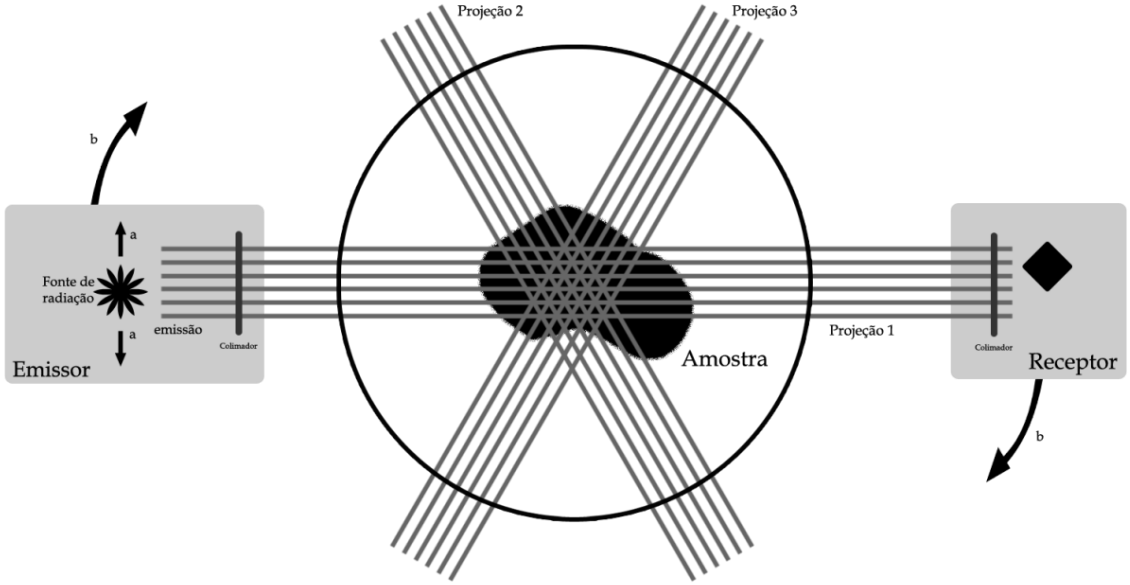

Figura 2 – Esquema básico de um tomógrafo

Na figura 2, temos uma visão mais detalhada do tomógrafo a ser simulado neste trabalho, de feixes paralelos (tomógrafo de primeira geração). O par emissor-receptor realiza 2 movimentos básicos: *translação*, indicado pela letra *a* na figura, e *rotação*, indicado pela letra *b*. A combinação desses movimentos forma uma malha como a vista na figura 3, onde é reproduzida a configuração de um tomógrafo 3x7.

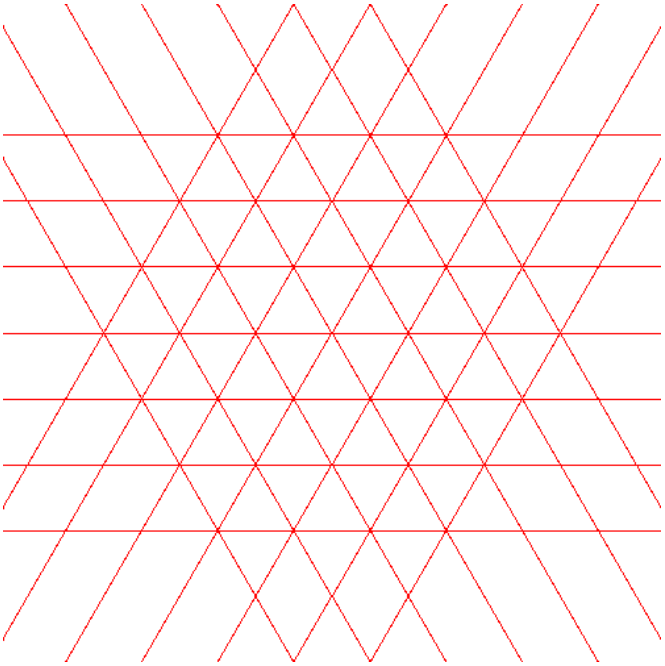

Figura 3 - Tomógrafo 3x7

2.2. TOMÓGRAFO – BASE FÍSICA E MATEMÁTICA

A atenuação sofrida por um raio eletromagnético em meio homogêneo é dada pela lei de Beer-Lambert:

$$I = I_0\, e^{-ux} \quad (1)$$

Em que I é a intensidade final do raio, I_0 a intensidade inicial ao sair do emissor, u o coeficiente de atenuação linear do material examinado e x o comprimento do segmento de reta pelo qual o raio atravessa o material.

Numa aplicação real típica, a amostra é composta de diferentes materiais, portanto de diferentes coeficientes de atenuação. Consequentemente, a equação torna-se um somatório:

$$I = I_0\, e^{\sum_i -u_i x_i} \quad (2)$$

Em que cada i representa um trecho x_i de material com coeficiente de atenuação u_i. Esta é a forma mais utilizada na prática para técnicas de reconstrução tomográfica computadorizada, por ser discreta. A solução contínua, por envolver integração, mostra-se ineficiente para este contexto.

3. PSIRT

O PSIRT foi desenvolvido como um algoritmo puramente sequencial (refs). Devido a sua natureza iterativa, com diversos loops aninhados e com dependências de dados diversas, ele não é trivialmente paralelizado. Abaixo, o pseudo-código do PSIRT "naive", isto é, sem otimizações e implementado de maneira a favorecer seu entendimento.

```
1.      // Atualize posições das partículas:
2.              Para cada partícula p,
3.                      força_resultante := 0
4.              Para cada trajetória t,
5.                      Calcule a distância d(p,t)
6.                      Calcule a força f(p,t) em razão da distância
7.                              força_resultante += f(p,t)
8.                      p.posição += força_resultante
9.      // Atualize forças das trajetórias:
10.             Para cada trajetória t,
11.                     t. n_partículas_atual := 0
12.                     Para cada partícula p,
13.                             Calcule a distância d(p,t)
14.                             Se d(p,t) < tolerância, faça:
15.                                     t.n_partículas_atual ++
16.                                     p.n_trajetórias ++
17.     // Cheque se convergiu:
18.             convergiu := 0;
19.             Para cada trajetória t,
20.                     Se t.n_partículas_atual >= t.n_partículas_convergência, faça:
21.                             convergiu ++
22.             Se convergiu == n_trajetórias, faça:
23.                     finalizou := verdadeiro
```

Código 1 - Pseudocódigo do PSIRT sequencial

Como pode-se ver, o algoritmo divide-se em 3 etapas:

- Atualizar posição das partículas
- Atualizar forças das trajetórias
- Checar convergência

Tal qual como descrito acima, porém, o PSIRT tende a um excesso de partículas quando estabilizado, dando origem à formação de ruído, i.e. partículas presentes na reconstrução sem correspondência com a amostra examinada. Isso ocorre quando uma partícula estabiliza-se num determinado ponto duma trajetória por onde não intersectam outras trajetórias. A interpretação física disto, conforme [3] explica, é que a partícula entra no cálculo de atenuação da trajetória a quem pertence, mas não se sabe exatamente em que ponto da trajetória ela se encontra. A solução é tentar fazer com que todas as partículas pertençam a interseções de trajetórias, como veremos a seguir. Nas figuras 4 e 5, vemos um fantoma e sua reconstrução pelo PSIRT segundo o pseudocódigo descrito acima.

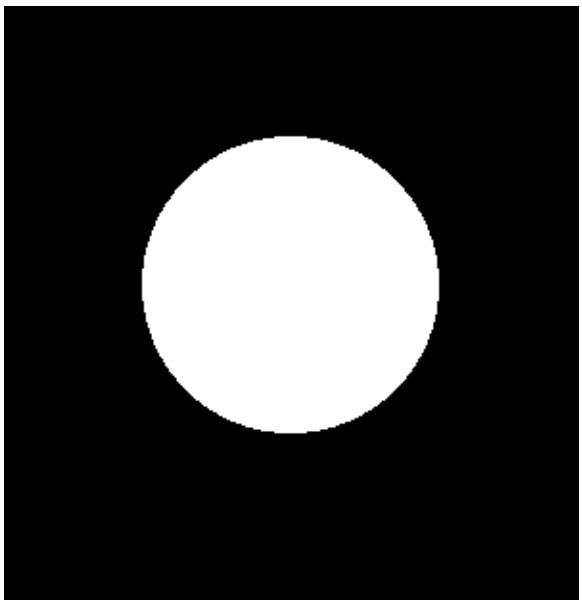

Figura 4 - Fantoma a ser reconstruído

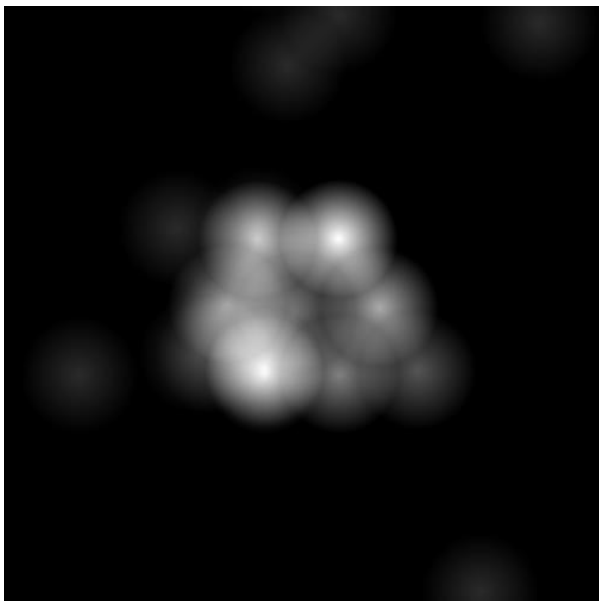

Figura 5 - Reconstrução sem otimização

3.1. OTIMIZAÇÃO

Ao adicionar uma etapa de otimização após a convergência, consegue-se remover boa parte do ruído mencionado na seção acima. A otimização consiste dos seguintes passos:

```
1.  Liste as partículas de acordo com a quantidade de trajetórias a que cada uma
    pertence, em ordem crescente
2.  Repita para todas as partículas listadas:
3.          Se a partícula não atenda a trajetória alguma, remova-a da lista
4.          Senão, marque a partícula
5.          Execute o PSIRT apenas com as partículas não-removidas e não-marcadas.
6.                  Se houve convergência após n ou menos iterações, desmarque a
                    partícula.
7.                  Se não houve, remova a partícula.
```

Código 2 - Pseudocódigo de otimização do PSIRT

Executando o algoritmo modificado com a otimização proposta, consegue-se remover parte significativa do ruído decorrente de partículas presas a trajetórias com poucas ou nenhuma intersecção. Na figura 7, vemos a reconstrução do fantoma da figura 6 sem a etapa de otimização, exibindo ruído na parte superior da imagem. Na figura 8, vemos a reconstrução do mesmo fantoma, mas com otimização.

Figura 6 – Fantoma a ser reconstruído

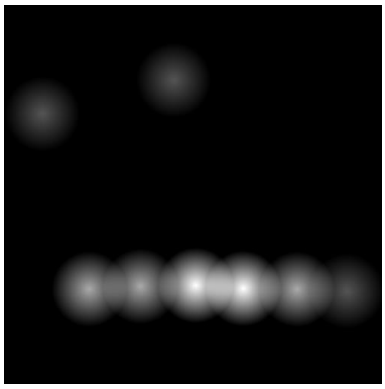

Figura 7 - Reconstrução sem otimização

Figura 8 - Reconstrução com otimização

3.2. PSIRT EM C

Afim de contar com um ponto de partida por onde iniciar a paralelização, decidimos por implementar o PSIRT em C. Além de servir de base para o algoritmo em CUDA-C, a implementação em C servirá de base comparativa nas medições de desempenho com o algoritmo paralelizado. O programa conta com leitor de sinograma em formato texto, renderizador em tempo real utilizando OpenGL e geração de bitmaps com a reconstrução tomográfica.

O programa foi compilado com o GCC 4.6.2. A implementação escolhida do OpenGL foi o FreeGLUT. Para o processo de profiling, utilizou-se o gprof.

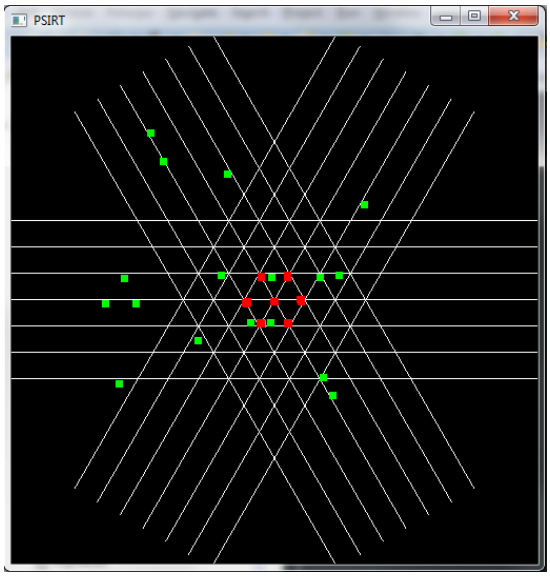

Figura 9 - Etapa de otimização do PSIRT

Na figura 9, vemos a execução do PSIRT durante a etapa de otimização. Cada partícula é representada como um ponto; pontos vermelhos são partículas *vivas* e pontos vermelhos, partículas *eliminadas*. A etapa de otimização consiste justamente em checar se cada partícula é imprescindível para a convergência do sistema: caso não seja, ela é eliminada, e na reconstrução apenas as partículas vivas são consideradas.

3.2.1. Profiling

Abaixo, trecho selecionado de resultados do programa de profiling *gprof*, executado sobre o algoritmo em C já mencionado. Para focarmos especificamente no algoritmo PSIRT, foram desativadas a renderização final da imagem reconstruída em bitmap e a renderização em tempo real por OpenGL.

```
Flat profile:


Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 22.00      0.22     0.22 23741931     0.01     0.01  magnitude
 21.00      0.43     0.21 11870964     0.02     0.05  distance
 17.00      0.60     0.17  5973828     0.03     0.04  directionFrom
  7.00      0.67     0.07   132846     0.53     2.77  update_trajectory
  6.00      0.73     0.06 11870964     0.01     0.01  normalize_void
  5.00      0.78     0.05 42593508     0.00     0.00  set
  5.00      0.83     0.05 23741928     0.00     0.00  minus_void
  5.00      0.88     0.05  5327346     0.01     0.01  trajectory_force
  4.00      0.92     0.04 12232124     0.00     0.00  sum_void
  3.00      0.95     0.03 23741928     0.00     0.00  dot_product
  3.00      0.98     0.03  5973828     0.01     0.10  resultant
  2.00      1.00     0.02 30000230     0.00     0.00  mult_constant_void
  0.00      1.00     0.00   284468     0.00     0.00  update_particle
```

Tabela 2 - Saída do gprof para PSIRT em C

As 3 funções que consumiram mais tempo de execução (60% do total) – magnitude, distance e directionFrom, destacadas em negrito – tem em comum o fato de serem chamadas pela função *resultant*, responsável por calcular a força exercida por uma trajetória sobre uma particular. No algoritmo apresentado, esta função tem de ser chamada, a cada iteração do PSIRT, uma vez para cada par partícula-trajetória, representando um grande gargalo de processamento. Portanto, as tentativas de otimizar e paralelizar o algoritmo terão de passar pela otimização desta função.

3.2.2. Discretização em pixels

Para se renderizar a imagem final recontruída a partir das partículas, decidimos pelo uso de pixels reais, i.e. cada pixel do bitmap é definido de acordo com sua distância às partículas. Apesar desta não ser a opção ideal quando se trata de RMSE [5], ela é adequada o suficiente ao foco deste trabalho, que é a performance. O objetivo da reconstrução no nosso contexto é apenas a verificação visual informal do resultado do algoritmo. Vê-se abaixo que, para esta finalidade, o método é suficiente: a figura 11 mostra o PSIRT reconstruindo o fantoma da figura 10. A figura 12 é o resultado final da execução do PSIRT com otimização.

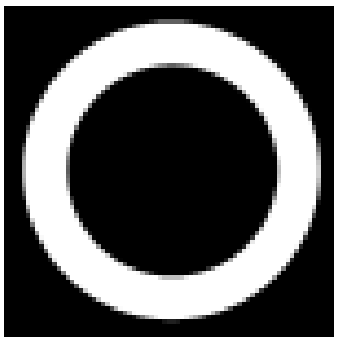

Figura 10 - Fantoma de um anel

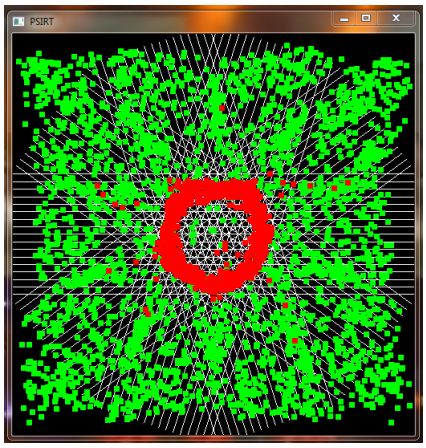

Figura 11 - Execução do PSIRT

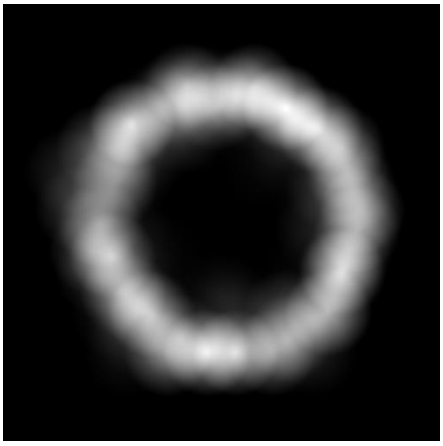

Figura 12 - Imagem reconstruída

4. CUDA

CUDA (*Compute Unified Device Architecture*) é uma plataforma e modelo de programação paralela para GPU criada e mantida pela nVidia. A linguagem de programação da plataforma[1], CUDA-C, é um subconjunto de C com algumas extensões.

Um software utilizando a plataforma CUDA consiste em 2 partes distintas: código *host* e código *device*. O código host é aquele em linguagem C, compilado por qualquer compilador C disponível no sistema operacional e executado na CPU. O código device é aquele em linguagem CUDA-C, compilado pelo nvcc (parte da SDK CUDA) e executado na GPU. Funções host são aquelas sem diretivas de pré-processamento (ou opcionalmente decoradas com __*host*__) e funções device são aquelas decoradas com __*device*__ ou __*global*__. Tradicionalmente, só a CPU pode invocar novas threads na GPU[2]. Dessa forma, o código host deve preparar todos os dados necessários ao algoritmo, fornecê-lo à GPU e invocar a função do device que cria e inicializa as threads da GPU. A essas funções dá-se o nome *kernel*, e elas são decoradas pela diretiva de pré-processamento __global__.

Enquanto uma típica CPU multi-núcleo tem 4 ou 8 núcleos, cada qual executando uma única thread, uma GPU com a arquitetura Kepler pode conter mais de 2 mil *CUDA cores* [6], cada qual executando mais de uma thread ao mesmo tempo. Como resultado, tem-se dezenas de milhares de threads concomitantes [7]. Além disso, o modelo de memória de uma GPU é radicalmente diferente de sua contraparte na CPU.

O desempenho de CPUs multi-núcleo modernas é altamente dependente do desempenho de suas caches, pois o custo para acessar a memória principal, mesmo com mecanismos como DMA, é alto: enquanto a cache L1 e L2 de cada núcleo tem largura de banda da ordem de 100GB/s, a memória principal tem banda de 10GB/s [8]. Dessa forma, caches grandes, de vários megabytes, fazem-se necessárias. Já uma GPU moderna, como as de arquitetura Kepler, tem memória principal de vários gigabytes e taxas de transferência da ordem de 200GB/s [6],

[1] Há também implementações em Fortran e C++.

[2] Placas gráficas mais recentes (compute capability 3.5) trazem um novo recurso, Paralelismo Dinâmico, que permite a invocação de kernels dentro de kernels, ou seja, permite à própria GPU iniciar novas threads.

reduzindo a necessidade de se ter grandes caches para cada núcleo, o que seria impraticável dado que existem milhares em cada GPU.

4.1. MODELO DE PROGRAMAÇÃO

Como mencionado, todo o código device reside na GPU, e o código host reside na CPU; tem-se portanto dois espaços de memória distintos. A comunicação entre esses dois espaços se dá pela função *cudaMemcpy*. Tipicamente, o host prepara os dados que serão usados pelo kernel, aloca um espaço equivalente na GPU, e transfere os dados para ela através do cudaMemcpy antes de começar o algoritmo. Após o kernel encerrar, o caminho oposto é feito para que a CPU possa exibir o resultado ao usuário[3].

O kernel, invocado por funções __global__, corresponde a um *grid* de execução. Cada grid é dividido em *blocos*, e cada bloco é composto por várias *threads*. Grupos de 32 threads são agrupados em *warps*, que são a unidade mínima de execução do agendador CUDA. As mais novas placas gráficas CUDA tem dois agendadores warp [9], o que permite o despacho simultâneo de duas instruções (uma de cada warp). Dentro de um grid, blocos executam paralelamente e tem sua própria memória individual. Cada bloco deve ter no máximo 512 threads; deve-se projetar o algoritmo de forma que cada bloco receba uma partição do problema e contribua para a resolução do mesmo.

[3] Uma nova funcionalidade disponível em GPUs com compute capability 3.5 e superior possibilita a comunicação direta entre GPU e outros dispositivos do computador, sem intermédio da CPU. Essa nova possibilidade pode significar o fim da necessidade de boa parte das transferências entre GPU e CPU.

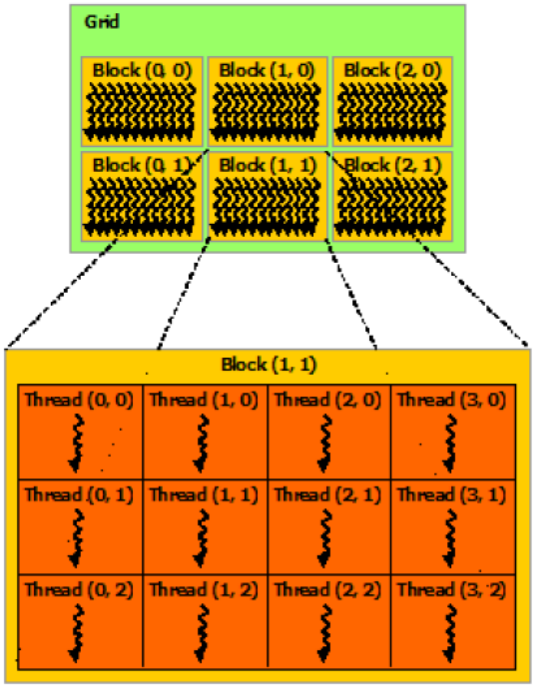

Figura 13 – Hierarquia de threads CUDA [10, p. 9]

De modo a facilitar a resolução de problemas com as mais diversas geometrias, CUDA permite até 3 dimensões para threads e blocos, como se vê na figura 13. Dessa forma, se o problema for bidimensional, pode-se interpretar o conjunto de blocos e de threads como matrizes; se o problema for tridimensional, pode-se interpretar os conjuntos como volumes. Essa característica é particularmente importante para se garantir a coalescência de memória.

Existem 3 categorias de memória disponíveis para grids, blocos e threads, descritos na tabela 3 e na figura 14.

Tipo	Visível a	Velocidade	Tamanho
Register	Thread	100x	1x
Shared	Bloco	10x	10x
Global	Grids	1x	100x

Tabela 3 – Hierarquia de memória CUDA

Memória global é aquela residindo na memória principal da placa gráfica e visível a todos os grids, com tamanho tipicamente de vários gigabytes. Memória shared é a memória que reside em cada multiprocessador, sendo portanto muito mais rápida que a global, mas limitada a 48 ou 64KB, dependendo da GPU. Cada multiprocessador tem ainda 32 ou 64 mil registradores de 32-bits [6]. Na figura abaixo, vemos o relacionamento entre as diferentes hierarquias de memória.

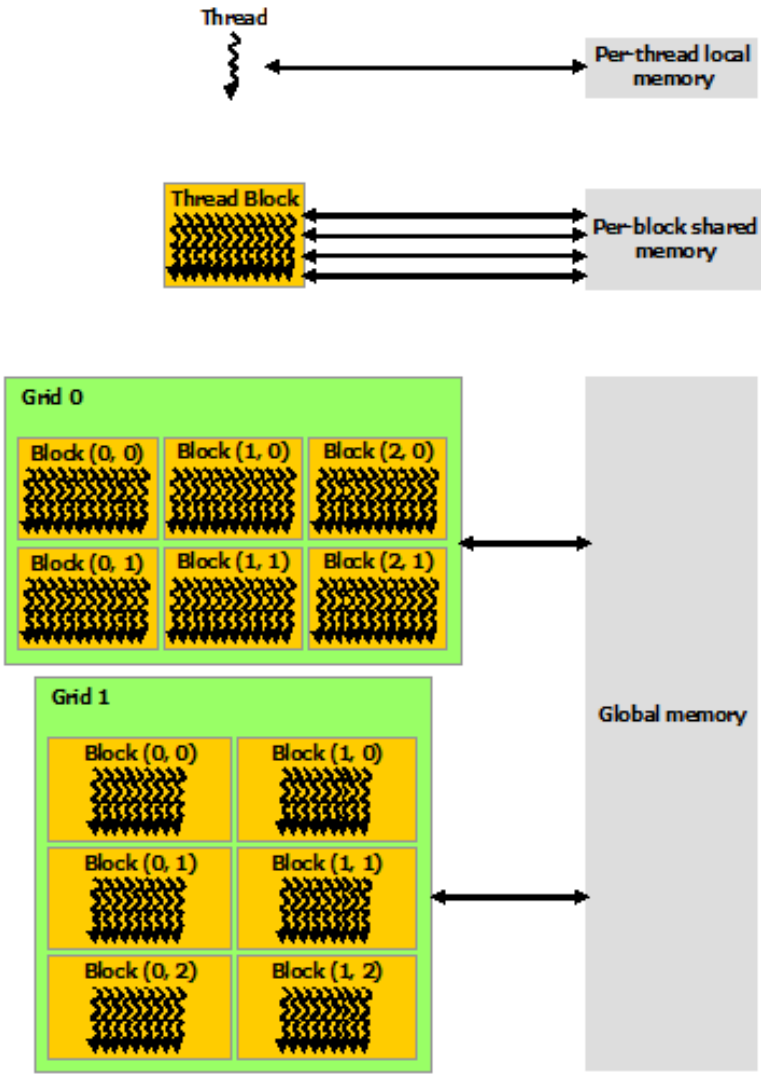

Figura 14 – Hierarquia de memória CUDA. [7, p. 11]

5. PARALLEL PSIRT (PPSIRT)

Como vimos no capítulo 5, para que se obtenha o melhor desempenho possível de um algoritmo iterativo em CUDA, ele deverá ser um conjunto de operações sobre dados coalescentes e compartilhados entre as threads, de modo que, idealmente, cada unidade do conjunto de dados seja tratado por uma única thread.

No caso do PPSIRT, há dois conjuntos de dados: o conjunto de partículas e o conjunto de trajetórias. O algoritmo exige a leitura e modificação dos valores de ambos a cada iteração. Além disso, não há qualquer relação de tamanho entre eles, i.e. os números de partículas e de trajetórias são completamente independentes. Este último fator impede que se utilize um único kernel para atingir o paralelismo em ambos os conjuntos.

Inicialmente, foi feita a opção por paralelizar apenas o conjunto das partículas, e serializar as operações sobre as trajetórias. Utilizou-se para tal uma única chamada para o kernel único, que continha o algoritmo inteiro e contava com um loop interno, o que evitava transferências entre host e device. Esta opção é justificável, pois normalmente há muito mais partículas que trajetórias, de modo que a atualização daquelas seja responsável por cerca de 72% do tempo de execução do algoritmo, contra apenas 28% gasto com atualização de trajetórias (ver seção *Profiling do PPSIRT*).

De modo a paralelizar o máximo possível do algoritmo, escolhemos outra abordagem: a divisão do algoritmo original em vários kernels diferentes. Essa metodologia permitiu atacar o problema de modo totalmente paralelo, pois cara kernel pode ter sua própria geometria. Em contrapartida, foi necessário utilizar um loop no host em vez de no kernel, e consequentemente transferências DeviceToHost a cada iteração (para checagem de condição de parada do loop). Abaixo, no código 3, o loop principal do PPSIRT, localizado no host:

```
while (ppsirt_status != STATUS_FINISHED)
{
        ppsirt_zero_traj<<<1, n_ttl_traj>>>(traj);
        ppsirt_update_traj<<<n_blocks, n_threads_per_block>>>(traj, part, d_ntraj);
        cudaDeviceSynchronize();
        ppsirt<<<n_blocks, n_threads_per_block>>>(traj, part, d_npart, ...);
        cudaDeviceSynchronize();
        ppsirt_chkstable<<<1, n_ttl_traj>>>(traj, d_tstable);
        cudaMemcpy(&ppsirt_status, d_st, sizeof(int), cudaMemcpyDeviceToHost);
}
```

Código 3 - Loop principal do PPSIRT, executado no host

Há 4 kernels com 2 diferentes parâmetros de execução responsáveis por executar o PPSIRT. Os dois kernels responsáveis por operar sobre as trajetórias executam um único bloco de tamanho igual ao número total de trajetórias do tomógrafo. Os kernels responsáveis por atualizar as partículas executam em vários blocos, de modo a tornar possível o uso de mais de 512 partículas.

5.1. DETALHAMENTO DOS KERNELS

5.1.1. ppsirt

É responsável pela atualização da posição das partículas e sua otimização. É o kernel "principal", responsável por encerrar o algoritmo quando finalizada a otimização das partículas.

Primeiro, atualiza-se as partículas conforme descrito no pseudocódigo do capítulo 3. Em seguida, sincroniza-se o bloco e tenta se adquirir o *lock* de otimização. A thread que consegue adquirir o lock prossegue para a otimização (mais sobre problemas de sincronização na seção *Sincronização do PPSIRT*).

5.1.2. ppsirt_chkstable

Determina se o algoritmo convergiu ou não. Checa em paralelo todas as trajetórias, adicionando atomicamente à variável global *stable* e sincronizando o bloco único ao fim com __syncthreads().

5.1.3. ppsirt_update_traj e ppsirt_zero_traj

Responsáveis por atualizar o número de trajetórias a que cada partícula atende e o número de partículas pertencentes a cada trajetória. Como ppsirt_update_traj segue a geometria do conjunto de partículas, i.e. possivelmente com vários blocos, é necessário sincronizar o device inteiro entre a chamada de ppsirt_zero_traj e ppsirt_update_traj, de modo a garantir que não haja condições de corrida.

5.2. HIERARQUIA DO PPSIRT

A configuração do PPSIRT, dividido entre vários kernels de diferentes geometrias, exige cuidado com sincronia entre threads, entre blocos e entre grids.

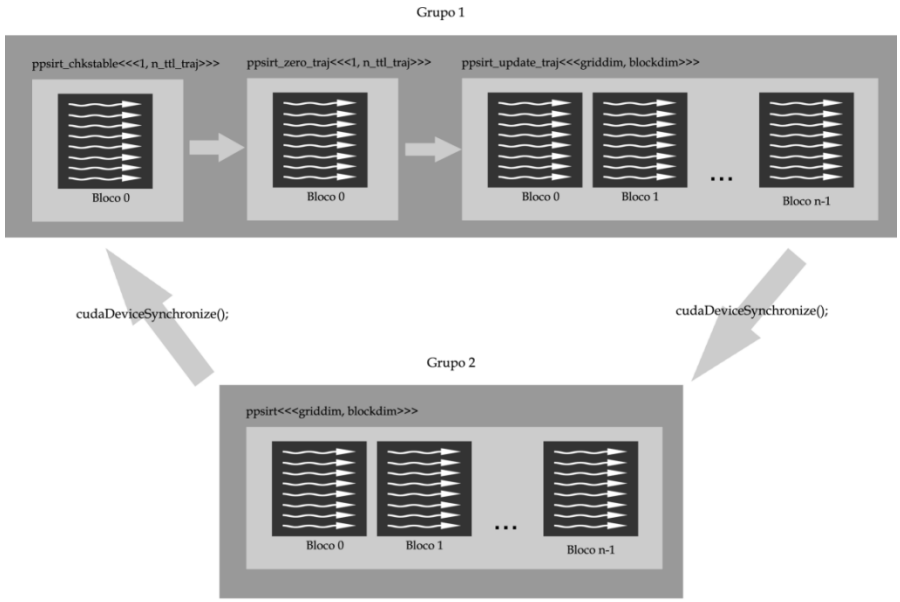

Figura 15 - Grids, blocos e threads executando em paralelo

Como vimos no capítulo anterior, ao se chamar dois kernels em sequência, o CUDART pode começar o segundo kernel sem esperar que o primeiro conclua. Dividimos portanto os 4 kernels do ppsirt em 2 grupos executados sequencialmente: o kernel ppsirt, que trata da atualização de partículas, executa sozinho num grupo; quando conclui, iniciam-se os outros 3 kernels. Não há condições de corrida: o kernel responsável por atualizar as trajetórias só inicia-se após as mesas serem zeradas por ppsirt_zero_traj (que possui um único bloco, sincronizado ao fim da execução), e a convergência só é checada em ppsirt_chkstable após as partículas serem todas atualizadas em ppsirt.

5.3. OTIMIZAÇÃO NO PPSIRT

A etapa de otimização impõe um desafio adicional à paralelização do PSIRT: como se vê pelo pseudocódigo da seção *3.1. – Otimização,* a operação de otimização consiste em checar sequencialmente o que acontece quando tenta se remover cada partícula. Não se pode simplesmente otimizar todas as partículas em paralelo simultaneamente, pois haveria condição de corrida; no pior caso, cada partícula tomada individualmente seria dispensável, e a otimização acabaria por remover todas as partículas. Foi necessária uma sincronização cuidadosa, como se vê na seção a seguir.

5.4. SINCRONIZAÇÃO DO PPSIRT

Como visto na seção 5.2., é preciso cuidado na sincronia entre as diferentes hierarquias de execução do PPSIRT. Além disso, o modelo de programação CUDA é suscetível também a perigos decorrentes de sincronia dentro dos kernels.

Na maior parte dos casos, assumindo que os blocos de um mesmo grid sejam independentes entre si, é possível resolver os problemas de sincronia apenas com a sincronização de cada bloco, com a função __syncthreads(). No entanto, a etapa de otimização exige maior cuidado.

Para garantir que uma e apenas uma partícula esteja em processo de otimização a qualquer dado momento, foi utilizado um *lock* implementado através da função atomicCAS

(compare and set atômico) e uma variável inteira. A cada iteração, a thread tenta adquirir o lock. Caso o lock esteja *destravado* e a partícula correspondente à thread não esteja marcada (i.e., ainda não foi otimizada), a thread trava-o, atomicamente preenchendo a variável com seu thread index. Em seguida, o processo normal de otimização segue como descrito no pseudocódigo da seção 3.2, com a diferença que no lugar de otimizar sequencialmente partícula por partícula, considera-se em otimização a partícula cuja respectiva thread possuir o lock. Assim, mantém-se o paralelismo: apesar de ainda otimizar uma partícula por vez, todas as demais threads seguem contribuindo para a otimização. A partícula detentora do lock é marcada como "em checagem" até que seja otimizada, e as demais partículas tem suas respectivas threads executando normalmente. Quando a partícula detentora do lock termina de ser otimizada, é marcada ora como "eliminada" ora como "checada", de modo que esta thread não irá tentar adquirir o lock novamente.

Além do lock de otimização, outros mecanismos de sincronização são utilizados no algoritmo. A variável *status* do ppsirt, que indica quando o algoritmo terminou de ser executado, é operada atomicamente com a função atomicCAS. A variável que armazena o número de partículas otimizada é incrementada só pela thread que possui o lock de otimização, e, além disso, é incrementada atomicamente. Todos os kernels fazem uso da função __syncthreads() para sincronizar seus blocos, e utilizam operações aritméticas atômicas quando necessário.

```
// so tenta pegar lock se nao foi otimizada
if (tid_optim_status != STATUS_OPTIMIZED &
!( (p[tid].status == CHECKED | p[tid].status == DEAD) )  ) {
        atomicCAS(optim_lock, OPT_UNLOCKED, tid);
}


if (*stable==*n_traj)
{
        atomicCAS(status, STATUS_RUNNING, STATUS_CONVERGED);
        if (*status == STATUS_OPTIMIZING && *optim_lock==tid) {
```

```
                p[tid].status = DEAD;
                atomicAdd(parts_optimized, 1);
                tid_optim_status = STATUS_OPTIMIZED;
                atomicCAS(status, STATUS_OPTIMIZING, STATUS_CONVERGED);
                atomicCAS(optim_lock, tid, OPT_UNLOCKED); // terminou de otimizar: liberar lock
        }
        else if (*status == STATUS_CONVERGED && *optim_lock==tid && p[tid].status == ALIVE) {
                        p[tid].status = CHECKING;
                        atomicCAS(status, STATUS_CONVERGED, STATUS_OPTIMIZING);
        }
}
else {
        if (*status == STATUS_OPTIMIZING && *optim_lock==tid) {
                if(++*optim_curr_iteration > MAX_ITER) {
                        p[tid].status = CHECKED;
                        atomicAdd(parts_optimized, 1);
                        tid_optim_status = STATUS_OPTIMIZED;
                        atomicCAS(status, STATUS_OPTIMIZING, STATUS_CONVERGED);
                        atomicCAS(optim_lock, tid, -1       }
        }
}



// checagem final para liberar lock
if (p[tid].status == CHECKED | p[tid].status == DEAD) {
        atomicCAS(optim_lock, tid, OPT_UNLOCKED);
}
```

Código 4- Trecho de otimização do kernel ppsirt

6. RESULTADOS

O PPSIRT foi comparado com a implementação em C, compilado com GCC, com profile gerado pelo gprof, como descrito no Capítulo 3. Para medição dos tempos de execução do PPSIRT, foi utilizado o nvprof, profiler incluso na SDK CUDA. Abaixo, um exemplo de output do nvprof, quando executado com o ppsirt:

Time(%)	Time	Calls	Avg	Min	Max	Name
72.01	38.22s	5694	6.71ms	3.27ms	7.97ms	ppsirt(Trajectory*, Parti...)
27.70	14.70s	5694	2.58ms	2.28ms	3.16ms	ppsirt_update_traj(Trajecto...)
0.26	138.55ms	5694	24.33us	20.74us	33.86us	ppsirt_chkstable(Trajecto...)
0.02	11.03ms	5695	1.94us	1.82us	8.96us	[CUDA memcpy DtoH]
0.01	6.77ms	5694	1.19us	1.10us	1.96us	ppsirt_zero_traj(Trajectory*)
0.00	35.17us	12	2.93us	736ns	14.43us	[CUDA memcpy HtoD]

Tabela 4 - Profile do ppsirt

Não é possível esperar por *speedups* lineares de acordo com o tamanho do problema, como é tradicional no processamento paralelo, dada a natureza do PSIRT/PPSIRT. A complexidade do problema depende tanto do número de partículas quanto da quantidade de projeções e trajetórias, além dos parâmetros diversos relacionados ao sistema de partículas. Além disso, é de se esperar também grande desvio padrão nas amostras por conta da aleatoriedade envolvida no algoritmo. Dito isto, obtemos melhoras significativas com o PPSIRT, especialmente quando utilizados sinogramas mais complexos e com maior número de partículas.

Ambos os algoritmos foram testados em um computador com a seguinte configuração de hardware:

CPU: Intel Core i7 3630QM, 2.4GHz, quad-core.

GPU: nVidia GeForce GT 635M, 2GB DDR3 dedicados, 96 cores CUDA.

Memória: 8GB.

Na figura 16, os três fantomas utilizados e suas respectivas configurações:

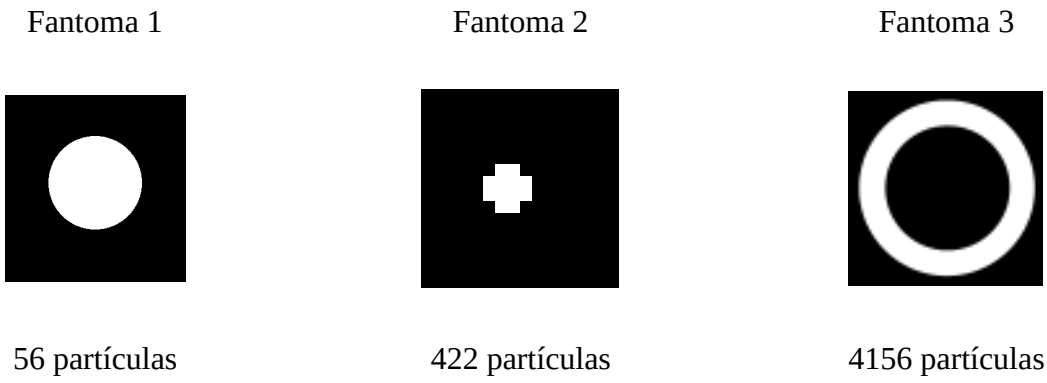

Figura 16 – Fantomas utilizados nos testes de desempenho

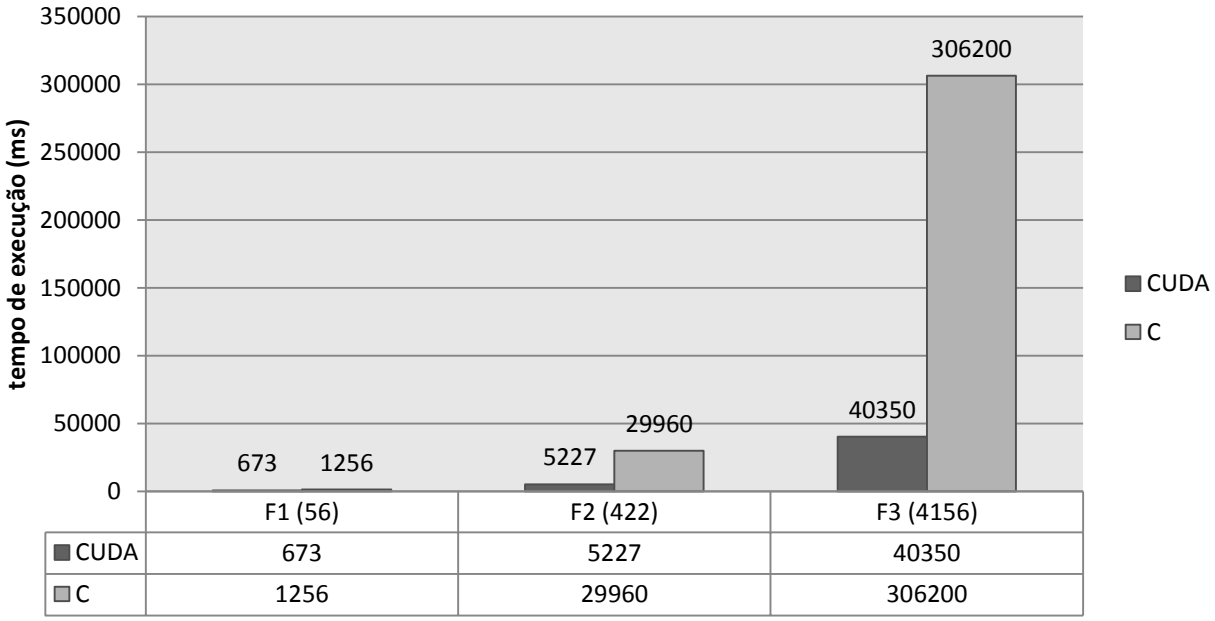

Figura 17 - Tempo de execução em milisegundos para os 3 fantomas

A figura 17 mostra a média aritmética dos tempos de execução para cada fantoma. Vemos claramente que a diferença entre os tempos de execução na CPU e em CUDA aumenta conforme aumenta a complexidade do fantoma. A figura 18 contém os mesmos dados da figura

17, utilizando porém escala logarítmica, de modo a explicitar os ganhos de desempenho nos diferentes fantomas.

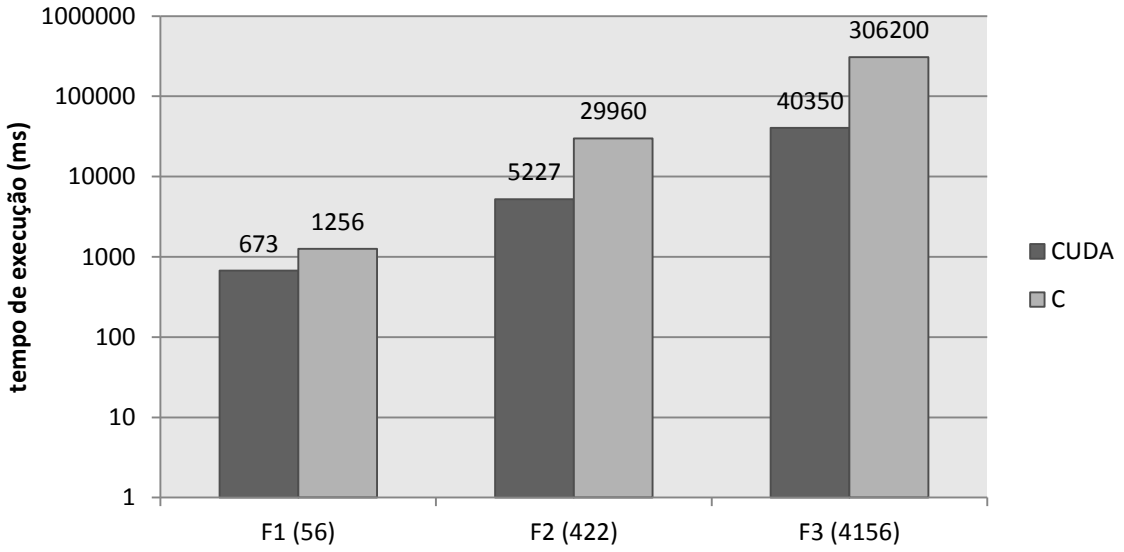

Figura 18- Tempo de execução para os 3 fantomas (escala log)

Tira-se o maior proveito possível da arquitetura CUDA quando se tem milhares ou dezenas de milhares de threads, o que explica a melhora no speedup conforme se aumenta a complexidade do fantoma - e consequentemente o grau de paralelismo – vista na figura 19. Especificamente, os speedups foram de 1,9x, 5,7x e 7,6x, para os fantomas 1, 2 e 3, respectivamente.

No entanto, o PPSIRT tem seu paralelismo limitado por necessitar de diversos mecanismos complexos de sincronização, o que impede uma melhora ainda maior no desempenho. Utilizando esta amostra, podemos prever aproxiadamente o speedup que poderia ser obtido em fantomas mais complexos, como vemos na linha tracejada do mesmo gráfico.

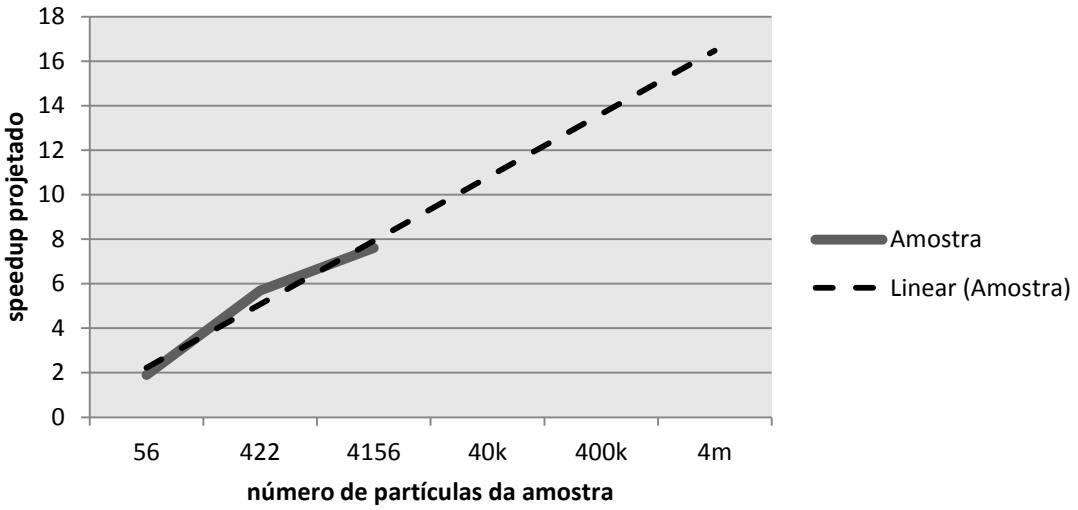

Figura 19 – Speedups mensurados e projetados

Como mencionado no começo deste capítulo, a natureza aleatória do sistema de partículas pode levar a resultados muito diferentes entre si. Comparamos, na figura 20, os melhores e piores casos de speedup. Definimos o melhor caso de speedup como a diferença entre a execução de pior desempenho da CPU e a execução de melhor desempenho da GPU, e, semelhantemente, o pior caso de speedup como a diferença entre a execução de melhor desempenho da CPU e a execução de pior desempenho da GPU.

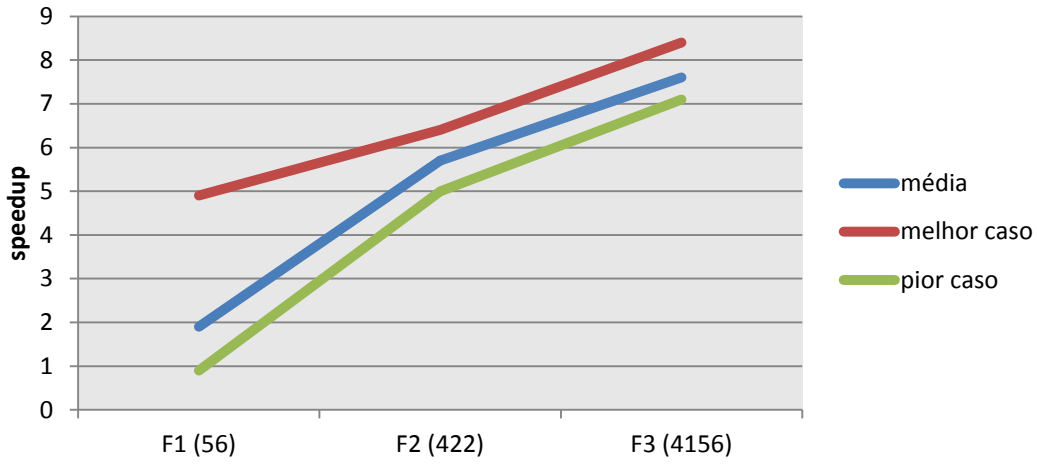

Figura 20 - Melhor caso, pior caso e média aritmética

Percebe-se maior consistência nos fantomas mais complexos. O fantoma de 56 partículas exibiu resultados bastante inconstantes; o melhor caso, no entanto, mostrou-se comparável ao dos demais fantomas.

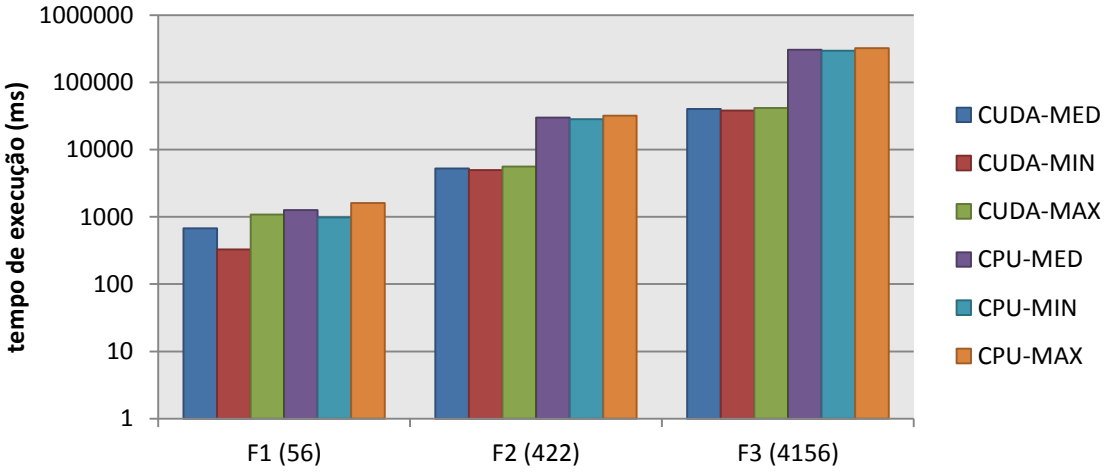

Figura 21 - Melhores e piores casos de CPU e GPU (escala logarítmica)

Percebe-se pela figura 21 mais uma vez a melhora de consistência nos tempos de execução. Se observarmos mais atentamente os casos do primeiro e último fantoma, podemos observar claramente a grande discrepância nos resultados do primeiro e a constância dos resultados do segundo. Nas figuras 22 e 23, temos gráficos detalhando os resultados desses fantomas

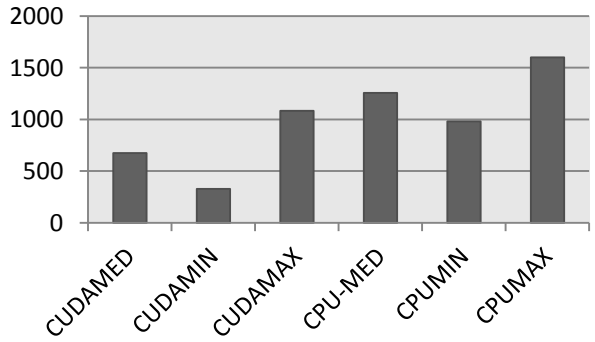

Figura 22 - Tempos de execução do fantoma 1

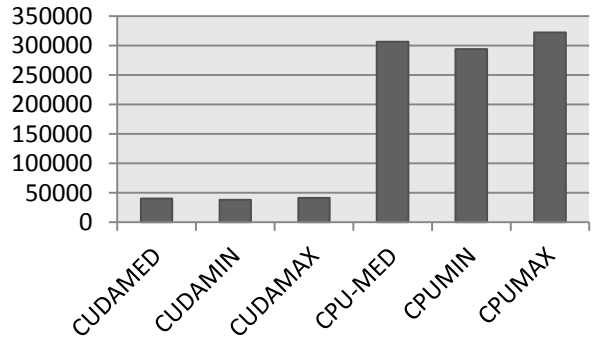

Figura 23 - Tempos de execução do fantoma 3

7. CONCLUSÃO

O PPSIRT, algoritmo implementado em CUDA-C e executado numa GPU, logrou ganhos de desempenho significativos quando comparado ao algoritmo original, PSIRT, implementado em C e executado numa CPU. Como é comum nos casos de algoritmos em CUDA-C, o speedup obtido varia bastante.

Os 3 fantomas testados, de 2 diferentes configurações (3x7 e 5x17) apresentaram speedups de 2x a 8x. Este resultado está dentro do esperado para algoritmos envolvendo sistemas de partículas [2], portanto, consideramos que o objetivo inicial do trabalho foi alcançado.

7.1. LIMITAÇÕES

O PPSIRT é limitado por dois grandes gargalos – as duas sincronizações globais do device no loop principal, chamadas a cada iteração do algoritmo. Estas sincronizações fazem-se necessárias por conta da interdependência entre as diferentes etapas do algoritmo executadas em kernels de diferentes geometrias.

Além disso, o desempenho também é tolhido pelas sincronizações dos blocos de cada kernel multi-bloco. A maneira ideal de eliminar essas limitações envolveria um novo algoritmo baseado no PSIRT onde se itera sobre estruturas de dados independentes e de geometria única.

Por limitações financeiras e de tempo, este trabalho limitou-se a testar o PPSIRT numa única GPU. Um estudo mais abrangente do PPSIRT demandaria o teste do algoritmo em diferentes GPUs, inclusive utilizando GPUs projetadas exclusivamente para uso de CUDA.

7.2. ESTUDOS FUTUROS

A evolução do PPSIRT passa necessariamente pela sua simplificação. O PPSIRT, por ser uma adaptação do PSIRT, não tem as características ideais para o uso na arquitetura CUDA. Um próximo passo natural seria o desenvolvimento de um novo algoritmo em CUDA-C utilizando os

conceitos do PSIRT e PPSIRT, mas que não seja necessariamente uma adaptação de nenhum dos dois.

Por ser uma tecnologia relativamente nova, CUDA ainda apresenta inovações constantes. Um novo recurso do CUDART chamado Paralelismo Dinâmico, disponível em algumas placas gráficas Quadro disponíveis no mercado, trás novas possiblidades para algoritmos CUDA: com ela, permite-se invocar kernels dentro de kernels, ou seja, permite-se à GPU criar novas threads.

REFERÊNCIAS

[1] Entertainment Software Association, 2013. [Online]. Available: http://www.theesa.com/facts/pdfs/ESA_EF_2013.pdf. [Acesso em 11 agosto 2013].

[2] nVidia Corporation, "Popular GPU-Accelerated Applications," [Online]. Available: http://www.nvidia.com/docs/IO/123576/nv-applications-catalog-lowres.pdf. [Acesso em 11 agosto 2013].

[3] S. MELO, Í. MOREIRA, C. DANTAS, G. JOHANSEN, B. HJERTAKER e R. MAAD, "The Particle System Iterative Reconstruction Technique in a High Speed Gamma-ray Tomograph," *6th International Symposium on Process Tomography,* 28 março 2012.

[4] R. A. KETCHAM e W. D. CARLSON, "Acquisition, optimization and interpretation of X-ray," *Computers & Geosciences,* p. 384, 2001.

[5] Í. MOREIRA, "Desenvolvimento do PSIRT para Tomografia Industrial," Recife, 2012.

[6] nVidia Corporation, "Inside Kepler," 2012. [Online]. Available: http://on-demand.gputechconf.com/gtc-express/2012/presentations/inside-tesla-kepler-k20-family.pdf. [Acesso em 11 agosto 2013].

[7] nVidia Corporation, "CUDA C Best Practices Guide," 2012, pp. 1-11.

[8] T. ROLF, "Cache Organization and Memory Management," 2009.

[9] nVidia, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. [Acesso em 11 agosto 2013].

[10] nVidia Corporation, “CUDA C Programming Guide,” 2012, p. 9.