Simulação de *tracking* da dinâmica dos elétrons em um acelerador síncrotron

Gustavo Ciotto Pinton[1] **- RA117136**

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Av. Albert Einstein, 1251, Cidade Universitária, Campinas/SP
Brasil, CEP 13083-852, Fone: [19] 3521-5838

gustavociotto@gmail.com

Abstract. *This report presents and describes the main aspects of the development of a parallel solution for a sequential algorithm, whose purpose is simulating the dynamics (modeled by a 6-element vector X) of electrons that run through a synchrotron accelerator. In general, in this algorithm, the whole extension of the accelerator is divided into several elements capable of changing the state of each electron in a certain way, and it is sought to determine which initial conditions of X will meet some conditions after a certain number of revolutions. All the parallelization was based on GPUs and the CUDA library, obtaining gains of performance of the order of 2500% compared to the serial execution.*

Resumo. *Este relatório apresenta e descreve os principais aspectos do desenvolvimento de uma solução paralela para um algoritmo sequencial de simulação da dinâmica do movimento dos elétrons (modelado por um vetor X de 6 posições) que percorrem um acelerador síncrotron. De maneira geral, neste algoritmo, divide-se toda a extensão do acelerador em diversos elementos capazes de alterar o estado de cada életron de uma determinada maneira, e busca-se quais condições iniciais de X atenderão algumas condições após um determinado número de voltas. Toda a paralelização foi baseada em GPUs e na biblioteca CUDA, obtendo-se ganhos de performance da ordem de 2500% em relação à execução serial.*

1. Introdução

Aceleradores de partículas síncrotron são constituídos, ao longo de todas suas circunferências, de atuadores, tais como dipolos, quadrupólos e sextupólos, responsáveis por modificar a direção dos elétrons através da imposição de campos magnéticos. Pode-se modelar o estado de um elétron na entrada do acelerador por um vetor no espaço de fase constituído por 6 elementos $X = (x_1, x_2, \ldots, x_6)$. A cada passagem por um determinado atuador, esse vetor é modificado por um mapa $F_n(X)$ que depende, evidentemente, do tipo de atuador e de seus parâmetros. A simulação de *tracking* da dinâmica de movimento dos elétrons consiste, portanto, em determinar quais vetores iniciais X_0 neste espaço de fase correspondem à órbitas estáveis após o percurso de N voltas pelo acelerador. Um vetor X_0 pode ser considerado uma órbita estável somente se, ao fim de N voltas completas, as posições $x_1, \ldots, x_6$ são inferiores às constantes $C_1, \ldots, C_6$. A explicação dos significados físicos de cada uma dessas posições não faz parte do escopo deste artigo.

Durante os experimentos, o acelerador foi modelado por um número constante $M = 10000$ de atuadores e a cardinalidade do conjunto de vetores iniciais testados foi igual a $I = 10000$. Além disso, variou-se o número de voltas N entre 10 e 10000, de modo a avaliar o ganho de performance em função do número total de iterações desenvolvidas.

Aproveitando-se a alta densidade de *cores* de arquitetura SIMD encontrada nas GPUs atuais, a paralelização desta simulação consistiu em distribuir o cálculo de cada condição inicial X_0 em uma *thread* distinta executada pela GPU, de modo a maximizar a quantidade de *threads* que rodam simultaneamente. A implementação foi a baseada no *toolkit CUDA*, desenvolvido para as placas de vídeo da *NVIDIA*. Como todos os vetores X_0 passarão pelos mesmos atuadores na mesma ordem e não farão acessos a posições não sequenciais de memória, a paralelização deste algoritmo não foi afetado pelos principais fatores que degradam a performance neste tipo de arquitetura, sendo eles, respectivamente, *branching divergence* e acessos *non-coalesced* à memória.

As próximas seções são dedicadas aos processos de implementação e aos resultados obtidos.

2. Análise da execução sequencial

Com o intuito de identificar qual trecho do programa possui o maior potencial de paralelização, a ferramenta `gprof` foi utilizada. Esse aplicativo é capaz de calcular o tempo de execução gasto para cada função, permitindo, assim, ao desenvolvedor uma visão global do desempenho do seu programa. Em essência, duas análises são oferecidas por ele, sendo elas, a *flat profile* e a *call graph*. A primeira mostra quanto tempo o programa gastou em cada função e quantas vezes tal função foi chamada. A segunda, por sua vez, permite, para cada função, visualizar quais outras funções ela chamou e por quais outras funções ela foi chamada. Essa análise pode ser útil para a determinação de trechos em que chamadas que tomam muito tempo podem ser economizadas.

A tabela 1 abaixo corresponde à fração mais relevante da análise *flat profile*, obtida através da execução da versão sequencial da simulação com $N = 10$.

Tabela 1. Análise *flat profile* da execução sequencial.

% time	cum. sec.	self sec.	calls	name
28.87	2.27	2.27	100000	DynamicSearch::**performOneTurn(double (&))**
15.77	3.51	1.24	1200000020	std::vector<RingElement>::**size()** const
15.58	4.74	1.23	400000000	Drift::**pass(double (&))**
14.12	5.85	1.11	400000000	Quadrupole::**pass(double (&)**
11.64	6.77	0.92	200000000	Sextupole::**pass(double (&)**
11.45	7.67	0.90	100000000	std::vector::**operator[]**(unsigned long)
...				

Observa-se, por meio dela, que 97.43% de todo o tempo gasto pelo programa está concentrado em apenas 6 funções. A análise do corpo da função `dynamical_aperture_search()` (contido no arquivo `DynamicSearch.cpp` e ilustrado pelo *psedo-código* 1, a seguir) explica perfeitamente a afirmação anterior, à medida que explicita três *loops* encadeados. Os dois primeiros são responsáveis por iterar sobre o conjunto de condições iniciais (de cardinalidade I), enquanto que o segundo, por

submeter cada uma das condições iniciais a N voltas pelos atuadores. A cada iteração do laço mais interior, realiza-se uma chamada à função `performOneTurn(double *)`, que, por sua vez, chama o método `pass` de cada um dos M atuadores. Desta forma, em linhas gerais, $I * N * M$ iterações serão executadas apenas neste trecho de código. É importante ressaltar que os dois *loops* responsáveis pela iteração sobre as condições iniciais podem ser classificados como *doall loops* e, portanto, podem ser paralelizados. Isso porque o processo de iteração de uma condição inicial $X_0^{m,n}$ é totalmente independente do processo associado à condição $X_0^{m,p}$, em que m, n e p representam índices dos *loops* que iteram sobre as condições iniciais. Em oposição, o laço mais interior e aquele presente na função `performOneTurn` são classificados como *doacross loops*, visto que as iterações executadas por eles dependem obrigatoriamente de resultados gerados por iterações anteriores. Essa afirmação é facilmente verificada em ambos os casos: para que a volta de índice i seja executada, ela necessita dos resultados que as de índices $i-1, i-2, \ldots, 0$ produziram, e para que o estado do életron $X_m^{i,j}$ seja calculado no atuador M_i, é necessário que os estados $X_{m-1}^{i,j}, X_{m-2}^{i,j}, \ldots, X_0^{i,j}$ associados aos atuadores $M_{i-1}, M_{i-2}, \ldots, M_0$ sejam conhecidos.

Algorithm 1 *Pseudo-código* do algoritmo sequencial

```
1:  function DEFAULT_DYNAMICSEARCH
2:      for (i, j) in 𝕀_x × 𝕀_y do
3:          X^{i,j} ← f(i, j)                ▷ Calcula condição inicial X^{i,j} em função de i e j
4:          for n ← 0 to N do     ▷ Itera sobre o número de voltas (performOneTurn())
5:              for m ← 0 to M do          ▷ Itera sobre o número de atuadores (size())
6:                  X^{i,j} ← g_{A[m]}(X^{i,j})  ▷ Muda X^{i,j} de acordo com atuador A[m] (pass)
7:              end for
8:          end for
9:      end for
10: end function
```

3. Paralelização

Esta seção é dedicada ao detalhamento do processo de implementação de um algoritmo paralelo baseado em CUDA e das principais dificuldades encontradas e como elas foram resolvidas.

3.1. Implementação

A fim de paralelizar a execução dos *doall loops* detectados na seção anterior, distribui-se a computação de cada condição inicial em uma *thread* distinta. Como discutido nesta mesma seção, isso pode ser realizado dado que tais cálculos são completamente independentes entre si.

CUDA distribui as *threads* em blocos unidimensionais, bidimensionais ou tridimensionais de até 1024 *threads*. Essa limitação existe devido ao fato de que todas as *threads* de um mesmo bloco residem no mesmo *core* e, desta forma, compartilham memória e tempo de processamento [NVIDIA 2017c]. A multidimensionalidade dos blocos garante que as *threads* sejam identificadas mais facilmente de acordo com a sua aplicação. No nosso caso, por exemplo, como as condições inicias são determinadas a partir de uma

tupla (i, j), utilizamos blocos de duas dimensões. Adicionalmente, tendo em vista que a quantidade máxima de *threads* por bloco é 1024, adota-se, neste artigo, blocos de 32×32. Assim como as *threads*, blocos também são distribuídos em *grids* de uma, duas ou três dimensões. Tendo em vista que a motivação para este tipo de divisão é a mesma que a explicada anteriormente, utilizaremos também *grids* bidimensionais com dimensões $\frac{I_x}{32} \times \frac{I_y}{32}$, em que I_x e I_y representam as dimensões do conjunto de condições iniciais.

Threads são executadas simultaneamente em blocos de 32, denominados *warps*. Tais blocos rodam em uma arquitetura SIMD, isto é, uma única instrução é capaz de realizar a mesma operação para todas as suas *threads*. *Warps* são executadas paralelamente por *streaming multiprocessors*, ou simplesmente *SMs*, cuja capacidade depende da *compute capability* da GPU. A escolha do número de *threads* por bloco também deve ser influenciada por essa capacidade, já que deseja-se maximizar o uso de um *SM*. Para a GPU *NVIDIA Tesla K40c*, de *compute capability* 3.5 e que foi utilizada nos testes, o número máximo de *warps* por *SM* é 64, totalizando, desta forma, 2048 *threads* que podem ser executadas simultaneamente [NVIDIA 2017b]. A escolha de blocos de 1024 *threads* maximiza, portanto, o uso de um *SM*, à medida que 2 blocos podem ser rodados paralelamente, ocupando, assim, toda a capacidade do respectivo *SM*.

Com a divisão de *threads* em blocos e *warps* já definida, a próxima etapa é implementar o *kernel* que será executado em cada uma delas e que calculará, a partir de uma condição inicial X_0, o respectivo vetor X_f após N voltas por M atuadores. Tal função recebe como parâmetro, além de N e M, o vetor A de `struct` contendo os atributos que descrevem cada um dos M atuadores. É importante ressaltar que, inicialmente, a ideia era transmitir um vetor de objetos que herdassem da classe abstrata `DynamicSearch` e que possuíssem as suas próprias implementações dos métodos virtuais. Porém, conforme discutido na subseção **Dificuldades**, esse tipo de operação não é suportado por *CUDA*. Adicionalmente, o *kernel* recebe também como paramêtro o vetor R, cujo propósito é armazenar os resultados calculados por todas as *threads*, de maneira que eles possam ser copiados para a memória da CPU posteriormente. Enfim, como a condição inicial depende dos índices (i, j) da *thread* em que ela será utilizada, ela pode ser calculada internamente e não necessita, portanto, ser transmitida ao *kernel* através de parâmetros. O pseudo-código 2, a seguir, exemplica as operações desenvolvidas por cada *thread* na GPU.

Algorithm 2 *Pseudo-código* do *kernel* que é executado pela GPU

```
1:  function DynamicSearchKernel(A, M, N, R)
2:      i ← blockDim.y * blockIdx.y + threadIdx.y        ▷ Índice y da thread
3:      j ← blockDim.x * blockIdx.x + threadIdx.x        ▷ Índice x da thread
4:      X^{i,j} ← f(i, j)          ▷ Calcula condição inicial X_{i,j} em função de i e j
5:      for n ← 0 to N do                                ▷ Itera sobre o número de voltas
6:          for m ← 0 to M do                            ▷ Itera sobre o número de atuadores
7:              X^{i,j} ← g_{A[m]}(X^{i,j})              ▷ Altera X^{i,j} de acordo com atuador A[m]
8:          end for
9:      end for
10:     R[i, j] ← X^{i,j}                                ▷ Armazena resultado calculado
11: end function
```

Vale lembrar que A e R devem ser alocados na memória do dispositivo através da chamada da função `cudaMalloc` e seus respectivos conteúdos devem ser copiados de/para a GPU através de `cudaMemCpy`. Ao fim, as memórias alocados são liberadas novamente por meio da função `cudaFree`.

3.2. Dificuldades

Duas principais dificuldades foram encontradas no desenvolvimento da solução paralela. Elas serão abordadas nas próximas subseções.

3.2.1. Objetos com métodos virtuais

Uma das principais classes que compõem o algoritmo sequencial é a classe abstrata `RingElement`, cujo principal propósito é fornecer uma base para a implementação de objetos capazes de representar um atuador. Na nossa aplicação, por exemplo, são fornecidas três classes que herdam diretamente de `RingElement`: `Drift`, `Quadrupole` e `Sextupole`. Tais classes fornecem uma implementação ao método virtual `pass()`, dependendo, evidentemente, de como influenciam na dinâmica de um elétron. O principal objetivo buscado ao empregar esse modelo de implementação é fornecer uma maneira simples e rápida para a adição de diferentes atuadores.

Assim como na solução sequencial, buscou-se utilizar essa abstração também na implementação de um *kernel* capaz de ser executado na GPU. A princípio, portanto, tal *kernel* receberia, como parâmetro, um vetor de objetos cujas classes herdassem de `RingElement` e, internamente, chamaria os métodos `pass()` corretamente de acordo com a classe derivada de cada elemento do vetor. Entretanto, a concretização desta proposta não foi possível, graças a uma limitação presente em *CUDA*. Segundo [NVIDIA 2017a], não é permitido transmitir um objeto de uma classe com métodos virtuais como parâmetro a uma função `__global__`. Isso porque, de acordo com este mesmo documento, a tabela de funções virtuais é colocada em uma posição de memória constante ou global pelo compilador `nvcc` e, desta forma, é inacessível internamente ao dispositivo.

A solução empregada a fim de contornar esse problema foi utilizar uma `struct` contendo todos os atributos de todas as classes derivadas de `RingElement`. Apesar do fato de que esta solução não otimiza o consumo de memória da GPU, sua simplicidade torna a adição de novos atuadores próxima da maneira original da solução sequencial.

3.2.2. Divergência das operações em ponto flutuante

Após que os primeiros resultados da solução paralela foram obtidos, sua comparação com aqueles da aplicação sequencial revelou que apenas um subconjunto bem reduzido dos vetores finais X_f divergia e, na maioria das vezes, essa divergência era encontrada a partir da terceira ou quarta casas decimais. Para $N = 10$ voltas, $M = 10000$ atuadores e um conjunto de entrada de cardinalidade $I = 10000$, por exemplo, dentre os 6100 vetores X_f encontrados pelas duas soluções, apenas um dentre eles divergia. A tabela 2, abaixo, exemplifica este caso. Para a compilação do programa sequencial, foi utilizada explicitamente a *flag* `-msse2`, instruindo o compilador a empregar a extensão SSE para o cálculo das operações em ponto flutuante.

Tabela 2. Vetores X_f divergentes entre as execuções paralela e sequencial.

	$X_f[0]$	$X_f[1]$	$X_f[2]$	$X_f[3]$	$X_f[4]$	$X_f[5]$
Serial	0.021950	0.000676	0.000370	0.000091	0.0	0.0
CUDA	0.022004	0.000684	-0.001117	-0.000063	0.0	0.0

Uma depuração profunda do código eliminou a primeira hipótese de que o *kernel* escrito não era compatível com a versão *serial*. A segunda hipótese, posteriormente confirmada, foi de que as pequenas variações observadas vinham do fato de que as operações de ponto flutuante fossem realizadas de maneira distinta entre a CPU e a GPU. Uma análise de [Whitehead and Fit-Florea 2011] determinou que, de fato, diferenças poderiam ser encontradas em relação ao conjunto de instruções SSE, utilizado pela CPU para este tipo de cálculo no exemplo acima. Segundo este artigo, GPUs com *compute capability* ≥ 2.0, como a que foi utilizada nos testes, suportam um tipo de operação denominada *fused-multiply-add* (ou simplesmente *FMA*) em pontos flutuantes de precisão simples e dupla, cujo principal objetivo é executar uma multiplicação e uma adição, isto é, computar $X * Y + Z$, através de uma única instrução. Sua principal vantagem, além do ganho em performance por executar duas operações de uma única vez, é que ela realiza um arredondamento a menos que uma operação de multiplicação seguida por uma de adição. Desta forma, denotando-se um arredondamento por $rn(x)$, o resultado de uma instrução *FMA* seria $rn(X*Y+Z)$, enquanto que o resultado de duas instruções, $rn(rn(X*Y)+Z)$. Evidentemente, a precisão após dois arrendodamentos é inferior a um apenas. Ao contrário do que ocorre no `nvcc`, a operação *FMA* não é habilitada por padrão no compilador `gcc` e o seu uso requer uma indicação explícita por meio de funções e *flags* específicas. Tal discordância foi justamente a fonte da divergência observada em uma pequena parte das soluções. Outra evidência constatada durante os testes e que confirma essa afirmação baseia-se no fato de que para N superiores, a quantidade de vetores X_f é superior. Isso significa que as diferenças geradas nos cálculos intermediários são propagadas e, quanto maior for o valor de N, mais elas serão potencializadas pelas iterações seguintes.

Os autores de [Whitehead and Fit-Florea 2011] afirmam que, para desabilitar o uso de instruções *FMA* na GPU, é necessário utilizar funções intrínsicas que informem ao compilador explicitamente para não unir uma multiplicação e uma adição. Neste caso, $X * Y + Z$ seria transformada na sentença `__dadd_rn(__dmul_rn(X,Y), Z)`. Aplicando-se este mesmo princípio à nossa aplicação, a fim de obtermos os mesmos resultados que a execução sequencial, foi necessário o uso destas funções em todas as sentenças que realizavam algum tipo de operação aritmética, mantendo, inclusive, a ordem dos operandos. Desta forma, a sentença `r[0] += length * r[1]` seria representada em CUDA por `__dadd_rn(r[0],__dmul_rn(length,r[1]))`, por exemplo. Em adição, o uso de instruções *FMA* também pode ser desabilitado através da *flag* `--fmad` do compilador `nvcc`. Caso ela seja modificada para 0 ou *false*, o compilador não gerará instruções que utilizem essa *feature*.

É importante ressaltar que nenhuma das duas soluções apresentadas acima está mais certa ou mais errada do que a outra. De fato, elas tratam-se apenas de tentativas de se aproximar de um valor cuja representação não pode ser fielmente realizada por meio de 32 ou 64 *bits*.

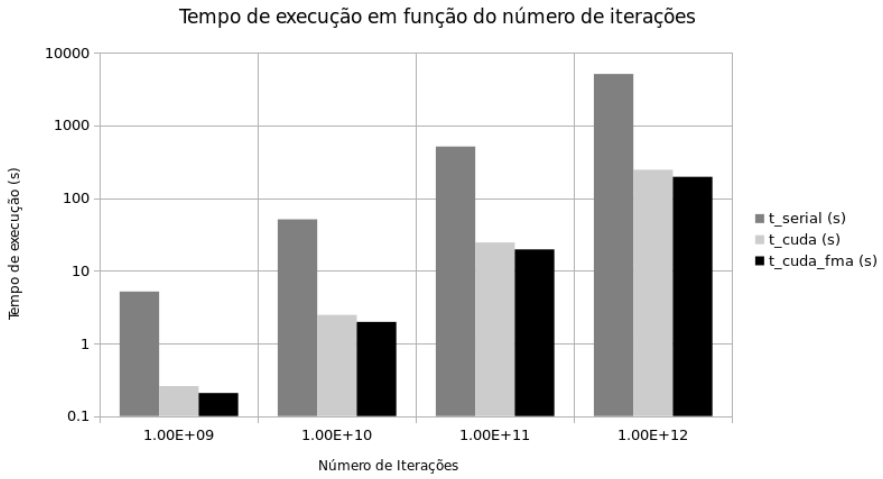

Figura 1. Tempos de execução para diversos valores de N

4. Resultados

Os resultados apresentados nesta seção foram gerados em um servidor com um processador *Intel(R) Xeon(R) E5-2620 v4* e com uma GPU *NVIDIA Tesla K40c* com *compute capability* 3.5. O código sequencial foi compilado com o compilador `gcc` com as *flags* `-O3`, `-Wall`, `--std=c++11`, `-mfpmath=sse` e `-sse2`. As duas últimas garantem que a extensão SSE será utilizada e que as operações em ponto flutuante sigam o padrão IEEE 754, tanto para precisão simples quanto para a dupla, assim como o compilador CUDA `nvcc` garante [Whitehead and Fit-Florea 2011]. Essas *flags* impedem que outras extensões sejam utilizadas, tais como a x87, que emprega precisão dupla-extendida com registradores de 80 *bits* com 64 deles dedicados somente à mantissa. [Monniaux 2008]. Operações realizadas nestes registradores produziriam resultados mais precisos, mas incompatíveis com os geradores com precisão dupla, que seguem estritamente o padrão IEEE 754.

Duas implementações paralelas foram realizadas, uma com suporte a instruções *FMA* e a outra compatível com a aplicação sequencial compilada conforme descrito acima. Para tal, foram propostas duas constantes, `CUDA_FMA` e `CUDA_INTRINSICS`, uma para cada implementação respectivamente. A escolha de qual delas deve ser utilizada é feita por meio da *flag* `-D` no comando de compilação.

A imagem 1 resume os resultados encontrados para as três execuções descritas acima com $I = 10000$, $M = 10000$ e quatro valores distintos de N, sendo eles 10, 100, 1000 e 10000. Verifica-se que o tempo de execução cresce proporcionalmente com o número de iterações, isto é, se o último cresce 10 vez, o outro também aumentará nesta mesma proporção. A figura 2, por sua vez, contém os *speedups* para cada uma das execuções. Em média, a solução que utiliza *FMA* tem um *speedup* de **26.821**, enquanto que a tradicional, **20.622**.

Observa-se que a melhor performance foi obtida empregando-se instruções *FMA* em CUDA. Conforme descrito na seção **Dificuldades**, além de uma melhor precisão, essa técnica também garante melhor desempenho, à medida que uma operação de multiplicação e outra de adição são realizadas por uma única instrução. Em oposição,

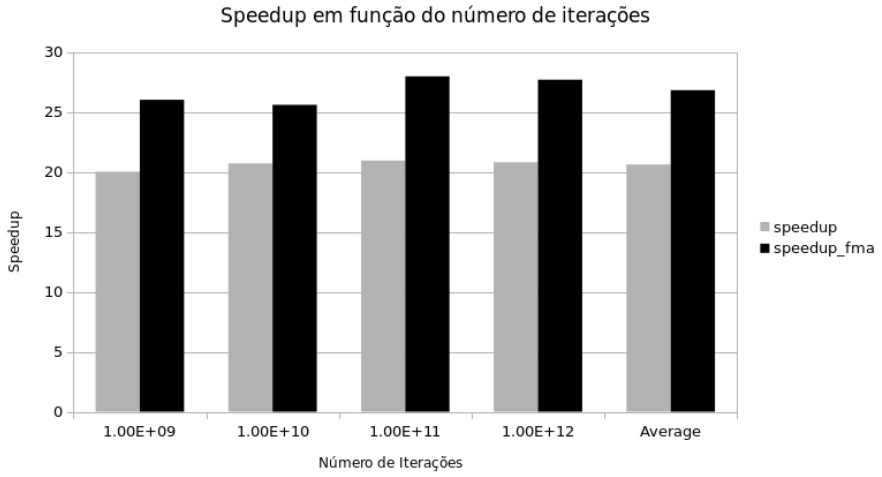

Figura 2. *Speedup* para diversos valores de N

ela não oferece exatamente os mesmos resultados que a simulação *serial*, dado o fato que ela realiza apenas um arredondamento a cada instrução *FMA*, ao contrário da solução tradicional, que produz dois arrendomentos para esta mesma operação.

5. Conclusões

Discutimos e avaliamos as principais dificuldades na paralelização, baseada em GPUs e CUDA, de um algoritmo de *tracking* da dinâmica de életrons submetidos em um acelerador síncrotron, . Dentre as principais dificuldades, destacamos a impossibilidade de transmitir a funções `__global__` objetos com métodos virtuais como parâmetro [NVIDIA 2017a] e as divergências de resultados entre CPU e GPU causadas por maneiras distintas de se efetuar operações de pontos flutuantes. Propomos, ainda, duas implementações para o algoritmo sequencial: uma que utilizasse instruções *fused-multiply-add* capazes de realizar a operação $X * Y + Z$ de *uma única vez* e a outra, semelhante ao método tradicional, isto é, que utizasse duas instruções separadas para esta mesma operação. Para a primeira, encontrados um *speedup* médio de **26.821** e, para a segunda, **20.622**.

Referências

Monniaux, D. (2008). The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(3):12.

NVIDIA (2017a). C/c++ language support. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#virtual-functions`.

NVIDIA (2017b). Compute capabilities. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities`.

NVIDIA (2017c). Cuda c programming guide. `http://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf`.

Whitehead, N. and Fit-Florea, A. (2011). Precision performance: Floating point and ieee 754 compliance for nvidia gpus. *nVidia technical white paper*.