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

Gustavo Ciotto Pinton - RA 117136


Universidade Estadual de Campinas - UNICAMP
MO644 - Introdução à Computação Paralela

24 de junho de 2017

# Introdução

- Conjunto de vetores $X_0^{i,j}$ representando estados de elétrons.
- Elétrons submetidos a $M$ atuadores em $N$ voltas.
- Simular quais vetores $X_0^{i,j}$ atenderão a determinadas condições ao fim de $N * M$ iterações.

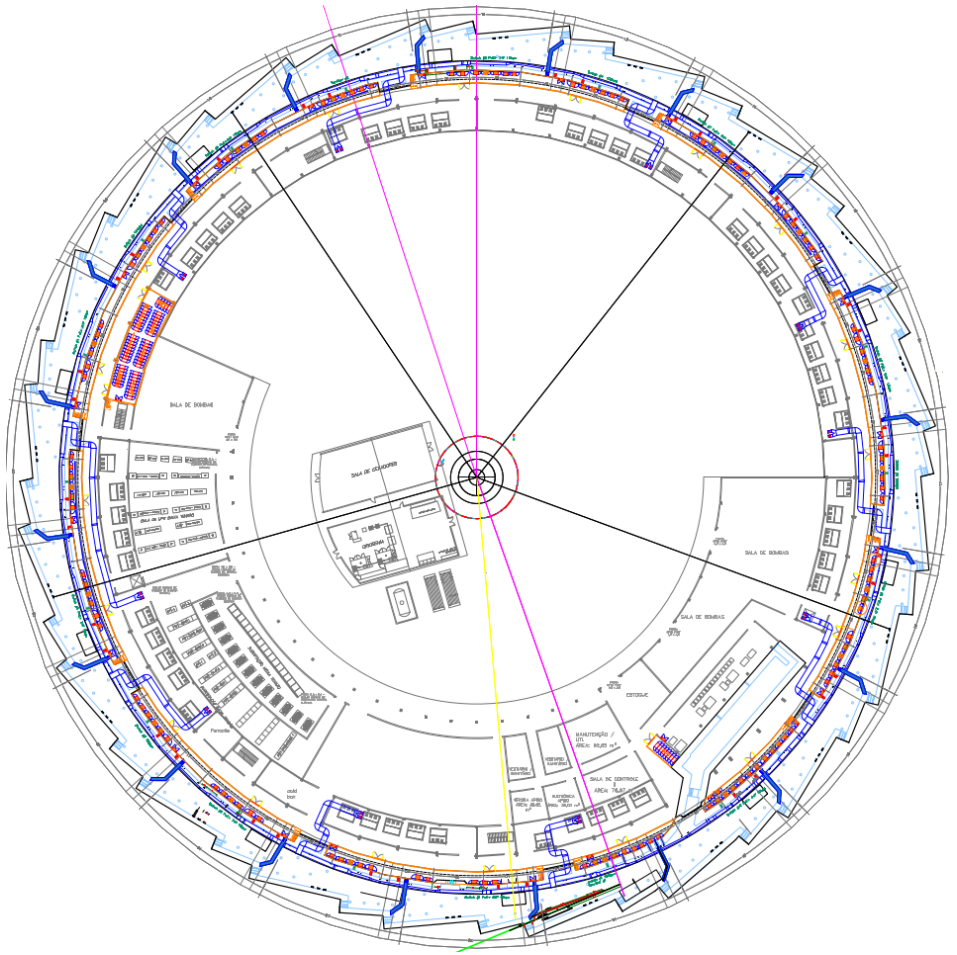

Figura: Acelerador *Sirius*

# Programa sequencial

*Profiling*

- Utilização do **gprof**.
- Análise *flat profile*: quanto tempo o programa gastou em cada função e quantas vezes tal função foi chamada.
- 97.43% de todo o tempo gasto pelo programa está concentrado em apenas 6 funções, todas chamadas dentro de `dynamical_aperture_search()`.

Tabela: 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) |
| ... | | | | |

# Programa sequencial

*Doacross e Doall loops*

- Laços externos são *doall*: uma condição inicial $X_0^{m,n}$ não depende de outra $X_0^{m,p}$.
- Laços internos são *doacross*:
  - Volta de índice $i$ necessita dos resultados que as voltas $i-1, i-2, \ldots, 0$.
  - Estado $X_m^{i,j}$ após atuador $M_i$ necessita dos estados $X_{m-1}^{i,j}, \ldots, X_0^{i,j}$ .

**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})   ▷ Altera X_i de acordo com atuador A[m] (pass())
7:              end for
8:          end for
9:      end for
10: end function
```

# Paralelização

## Implementação

- Distribui-se a computação de cada condição inicial $X_0^{m,n}$ em uma *thread* distinta: paralelização dos *doall loops*.
- Cada *thread* executa os *doacross loops*.
- Blocos de dimensões $32 \times 32$, visto que a quantidade máxima de *threads* por bloco é 1024.
- Maximizar o uso de um SM: para a *NVIDIA Tesla K40c*, de *compute capability 3.5*, o número máximo de warps por SM é 64, totalizando 2048 threads.
- Dois blocos inteiros por *SM*.

# Paralelização

## Implementação

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

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

# Paralelização

## Resultados

- Testes no *parsusy*: *Intel(R) Xeon(R) E5-2620 v4* e *NVIDIA Tesla K40c*.
- *Flags* `gcc`: `-O3`, `-Wall`, `-std=c++11`, `-mfpmath=sse` e `-sse2`.
- Duas implementações paralelas realizadas: uma com suporte a instruções *FMA* (um arredondamento) e a outra compatível com a aplicação sequencial (dois arredondamentos).

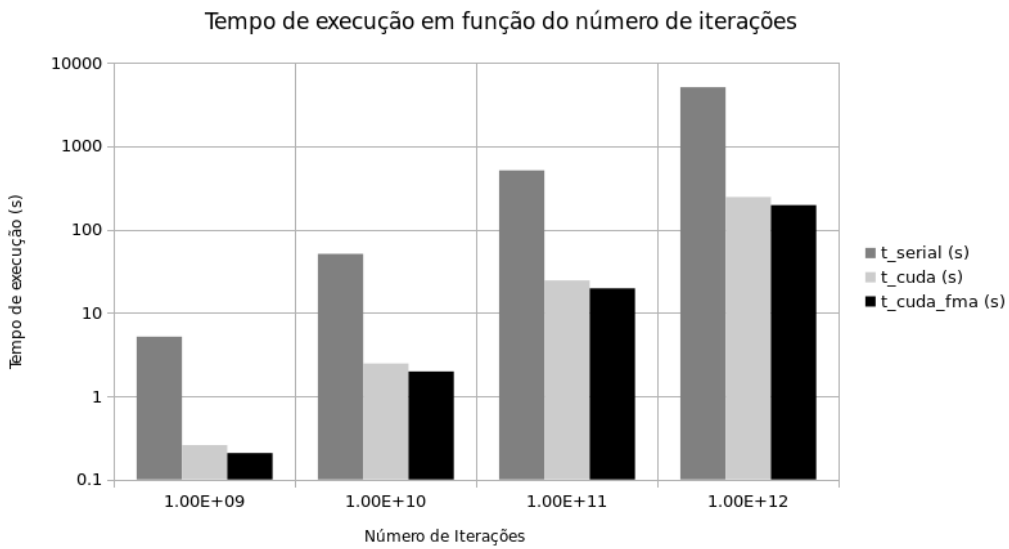


Figura: Tempos de execução para diversos valores de *N*

# Paralelização

## Resultados

- Em média, a solução que utiliza *FMA* tem um *speedup* de **26.821**, enquanto que a tradicional, **20.622**.

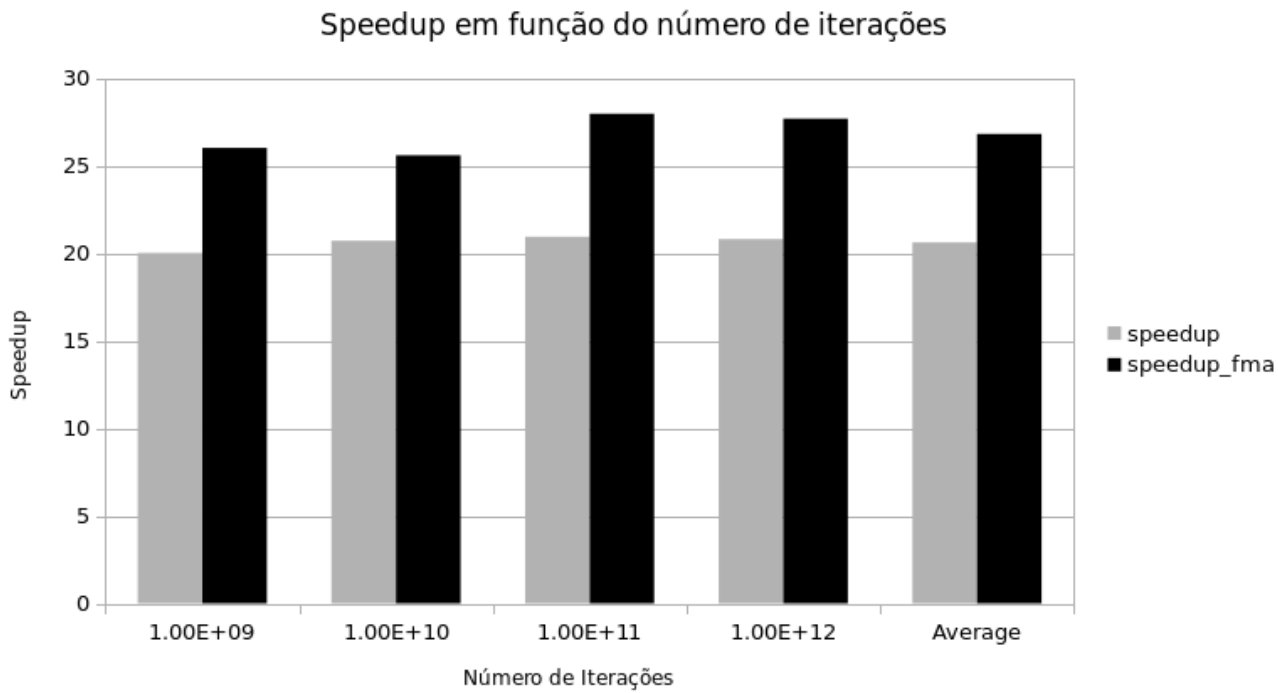


Figura: *Speedup* para diversos valores de *N*

# Paralelização

## Dificuldades

- *It is not allowed to pass as an argument to a* `__global__` *function an object of a class derived from virtual base classes.* [NVIDIA. C/C++ language support]
- A tabela de funções virtuais é colocada em uma posição de memória constante ou global pelo compilador `nvcc`.

- GPUs com *compute capability* $\geq 2.0$: operação denominada *fused-multiply-add* (*FMA*) em pontos flutuantes de precisão simples e dupla.
- Executar uma multiplicação e uma adição, isto é, computar $X * Y + Z$, através de uma única instrução.
- Ganho em performance por executar duas operações de uma única vez.
- Realiza um arredondamento a menos que uma operação de multiplicação seguida por uma de adição.
- $rn(X * Y + Z) \neq rn(rn(X * Y) + Z)$.

# Referências


- Monniaux, D. (2008). The pitfalls of verifying floating-point computations. ACM Transactions on Programming Languages and Systems (TOPLAS).
- NVIDIA. C/C++ language support. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#virtual-functions`.
- NVIDIA. Compute capabilities. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities`.
- NVIDIA. 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 NVIDA GPUs. nVidia technical white paper.