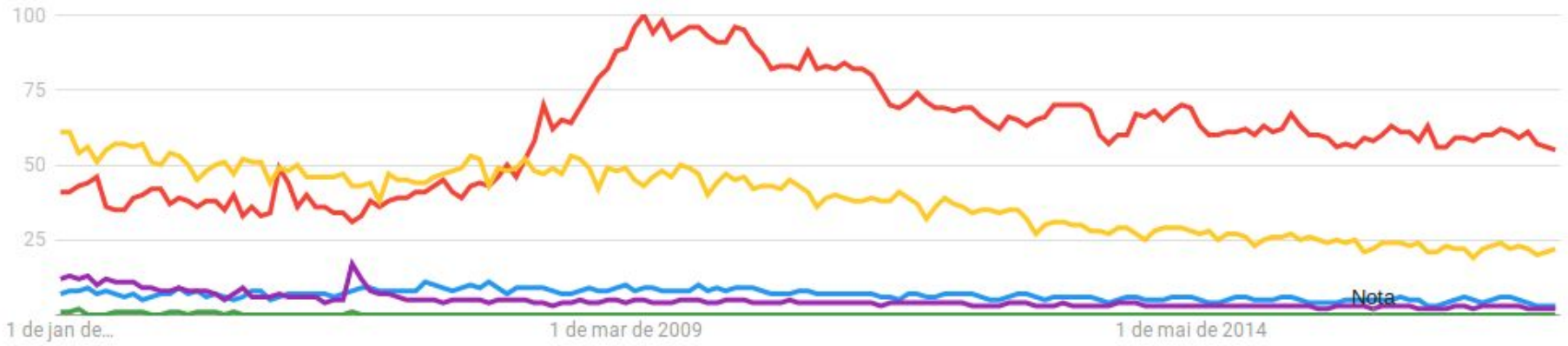
Programação de Arquiteturas Paralelas

Clícia Pinto

Google Trends

- CUDA
- OpenMP
- Parallel Computing
- HPC
- Heterogeneous Computing



O curso

A proposta desta aula é apresentar **aspectos introdutórios** de **programação** para plataformas de **alto desempenho** baseadas em **CPU/GPU**.

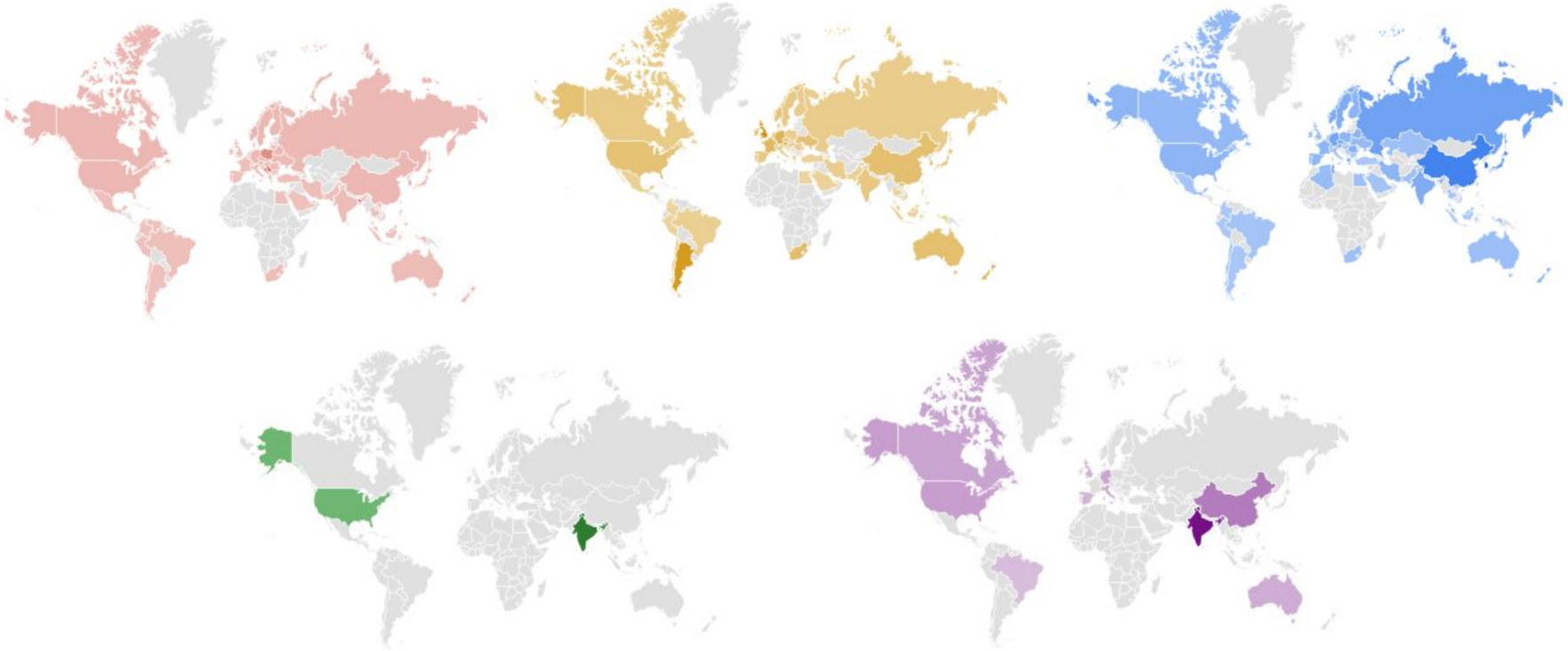
CUDA

OpenMP

Parallel Computing

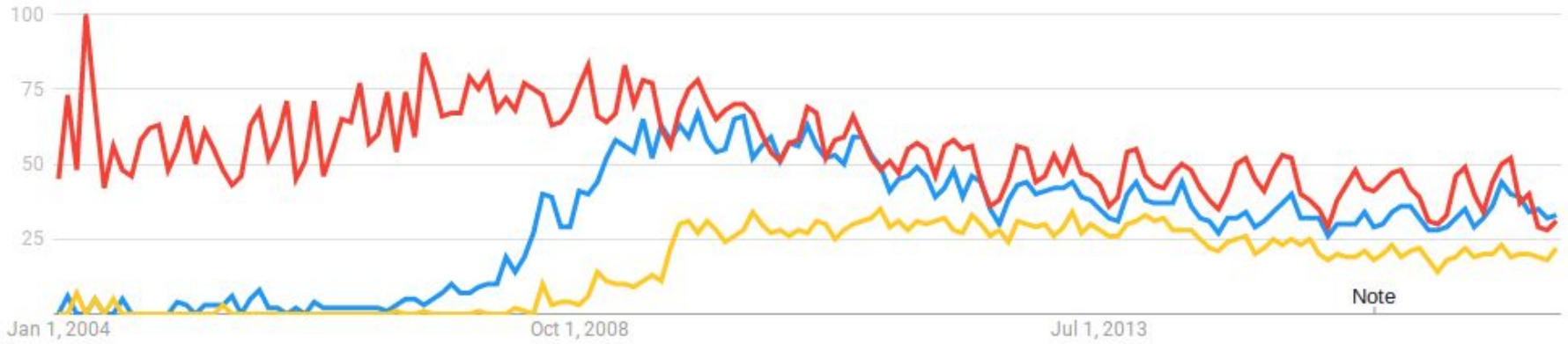
HPC

Heterogeneous Computing



Google Trends (Science)

OpenMP CUDA OpenCL



Métricas de avaliação do desempenho

- Speedup

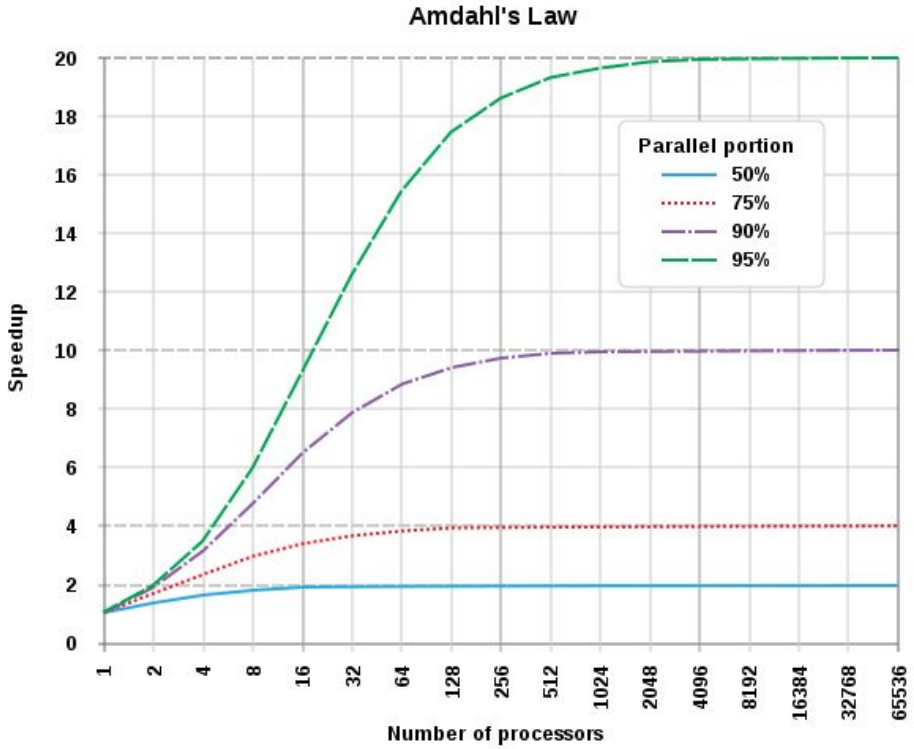
$$S(n) = \frac{\text{tempo de execução serial}}{\text{tempo de execução paralela}} = \frac{t_s}{t_p}$$

- Mensuração do ganho entre o código paralelo vs. o melhor código sequencial.
  - Exemplo: Multiplicação de Matrizes
    - 180 segundos (sequencial)
    - 30 segundos (paralelo)
    - Speedup de 6x

Métricas de avaliação do desempenho

- Speedup - Lei de Amdhal (1967)
  - Estabelece um limite superior para o speedup de um algoritmo paralelo

$$S(N) = \frac{1}{(1 - P) + \frac{P}{N}}$$



Métricas de avaliação do desempenho

- O pintor de cercado
  - Preparo da tinta = 30s
  - Pintura das estacas = 300s
  - Tempo para a tinta secar = 30s

- **Tarefas Sequenciais?**
- **Tarefas Paralelizáveis?**

Métricas de avaliação do desempenho

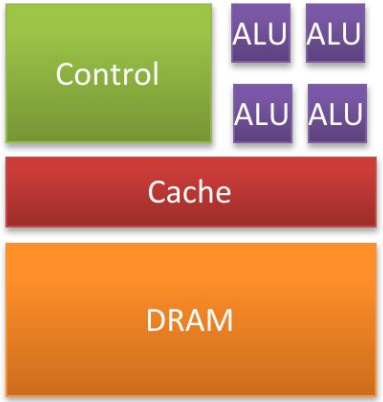
Pintores	Tempo	Speedup
1	30 + **300** + 30 = 360	1.0
2	30 + **150** + 30 = 210	1.7
10	30 + **30** + 30 = 90	4.0
100	30 + **3** + 30 = 63	5.7
∞	30 + **0** + 30 = 60	6.0

GPU+CUDA

Programação em GPUs

- O processamento gráfico caracteriza-se por uma computação massiva.
- O processamento gráfico é intrinsecamente muito paralelizável.
  - Muitas operações são feitas através da aplicação de um paralelismo com granularidade fina

Arquitetura das CPUs vs GPUs



- Grandes **caches**
- Poucos elementos de processo
- Otimização das Localidades Espacial e Temporal
- Sofisticados sistemas de controle

- Caches reduzidas
- Elevado número de elementos de processo
- Acesso otimizado para acesso sequencial de dados (streaming)

Como programar para GPUs?

- CUDA Extensão do C, C++, Fortran
- Objetivo
  - Escalabilidade em 100’s de cores, 1000’s threads paralelas
  - Foco no desenvolvimento de algoritmos paralelos
  - Possibilita computação heterogênea (CPU+GPU)
- Define
  - Modelo de Programação
  - Arquitetura

Modelo de Execução

- Modelo de Programação SIMD.
- A unidade de execução chama-se **kernel**
  - O paralelismo se baseia em threads.
  - O kernel de execução consistirá em um conjunto de threads.

Modelo de Execução

- Funções que executam na GPU podem ser declaradas com dois modificadores:
  - __global__
    - Funções que são invocadas à partir do host
  - __device__
    - Funções que são invocadas à partir do device

Modelo de Execução - Função Main

1) Declara as variáveis que serão utilizadas no Host
2) Aloca memória para as variáveis utilizadas no Host
3) Declara as variáveis que serão utilizadas no Device
4) Aloca memória para as variáveis utilizadas no Device
5) Envia os dados do Host para o Device
6) Define a dimensão do grid e dos blocos
7) Invoca o kernel paralelo
8) Envia os dados do Device para o Host
9) Libera a memória da GPU e da CPU

Executando kernels

- Chamada de função modificada:

```
kernel<<<dim3 grid, dim3 block>>> (...)
```

- Exemplo:

```
dim3 grid(32)
dim3 block(512)
kernel<<<grid, block>>> (...)

kernel<<<32, 512>>> (...)
```

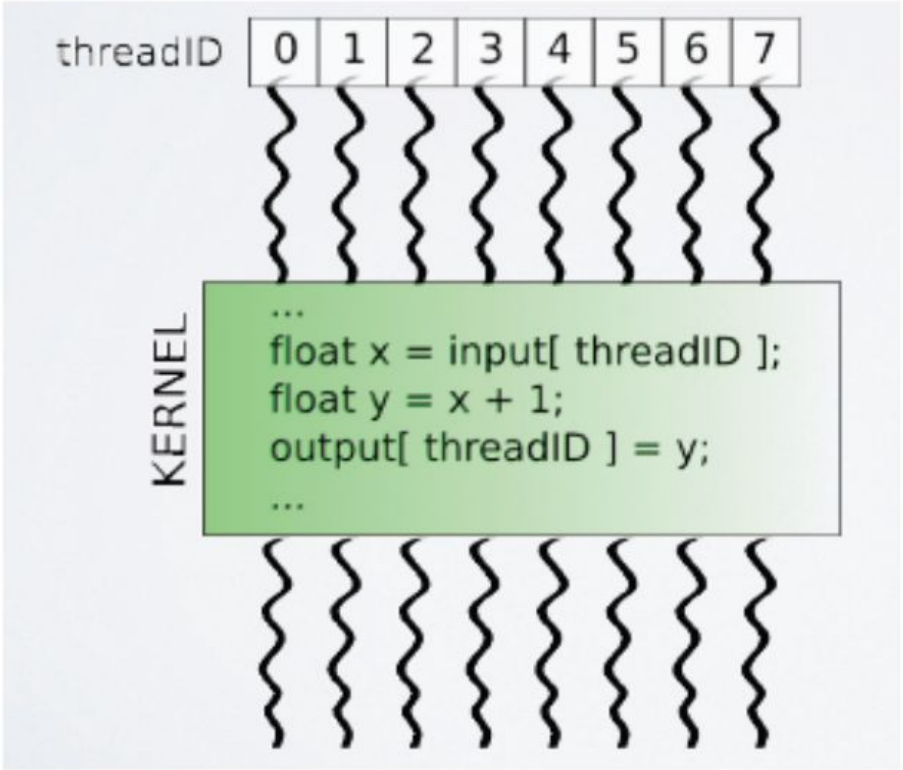
Modelo de Execução - Kernel

1) Um kernel em GPU lança a quantidade de threads, blocos e grids especificada no host
2) Todos os threads executam o mesmo código
3) Considerando uma execução 1D:
    a) blockIdx.x define o Id do bloco
    b) threadIdx.x define o Id da thread (dentro de um bloco)
    c) blockDim.x define a dimensão do grid

Executando kernels

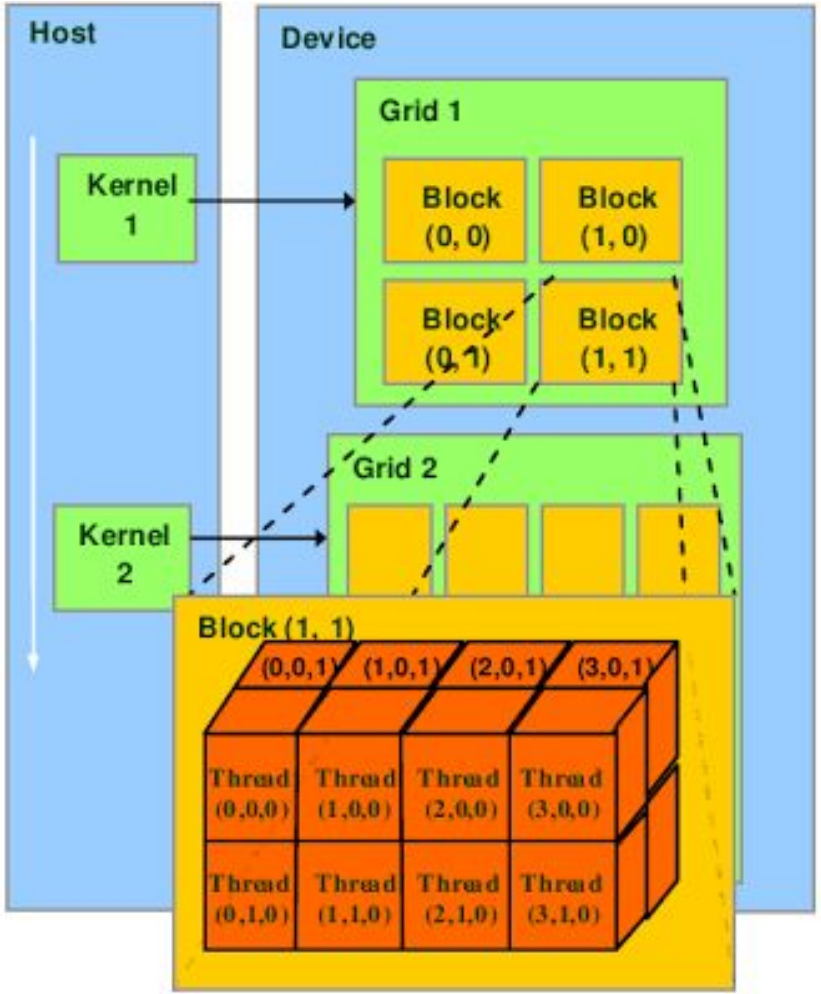
```
__global__ void kernel(int *dev_a, int value){
    int idx = blockDim.x * blockIdx.x + threadIdx.x
    dev_a[idx] = value;
}
```

Modelo de Execução

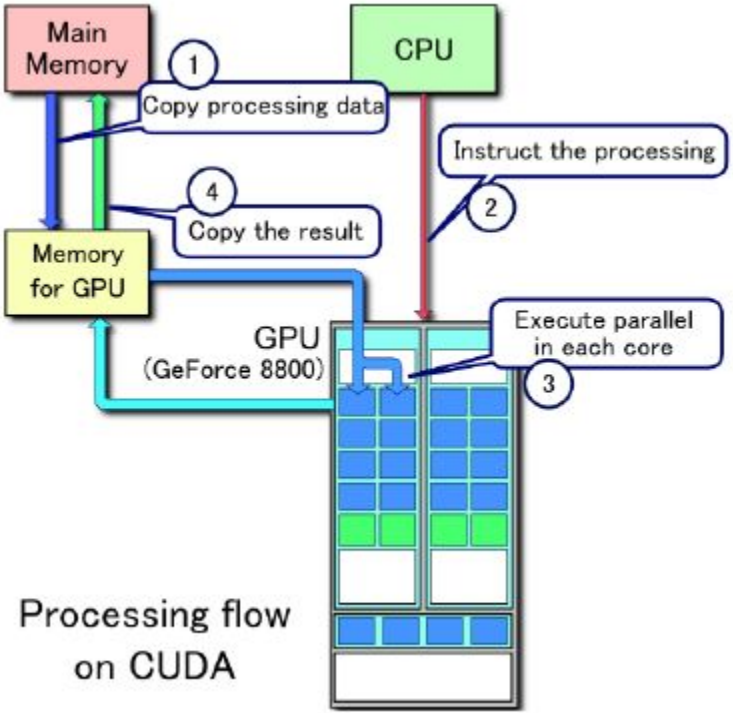


Modelo de Programação

- Cada thread tem associado:
  - Identificador próprio dentro do bloco (1D, 2D o 3D)
  - Identificador do bloco que pertencem dentro do grid



Fluxo de Execução



Modelo de Execução

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <cuda.h>

__global__ void kernel( int a, int b, int *c ) {

   *c = a + b;
}

int main( void ){
   int c;
   int *dev_c;
   cudaMalloc( (void**)&dev_c, sizeof(int));
   kernel<<<1,1>>>( 2, 7, dev_c );
   cudaMemcpy( &c,dev_c, sizeof(int),cudaMemcpyDeviceToHost ) ;
   printf( "2 + 7 = %d\n", c );
   cudaFree( dev_c );
   return 0;
}
```

Transferências de dados

- Chamada desde o Host
  - **`cudaMemcpy`**`(void * dst, void * src, size_t nbytes, enum cudaMemcpyKind direction)`
- API
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToHost`
  - `cudaMemcpyDeviceToDevice`

Gestão da Memória

- O host (CPU) realiza a reserva e liberação de memória
  - **cudaMalloc**(void ** ptr, size_t nbytes)
  - **cudaMemset**(void * ptr, int value, size_t count)
  - **cudaFree**(void * ptr)

```
int main(int argc, char **argv){
  int n = 1024 * sizeof( int );
  int * d_A = NULL;

  cudaMalloc( (void **)&d_A, n );

  cudaMemset( d_A, 0, n );

  cudaFree( d_A );
}
```

Espaços em Memória

- CPU e GPU possuem espaços de memória independentes
  - Os dados se movem através do ‘bus PCIExpress’
  - Reserva/Cópia/Liberação explícitas
  - Qualquer operação Reserva/Cópia/Liberação é realizada pelo host

Modelo de Execução - Soma de Vetores

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <cuda.h>
#define N 16

__global__ void kernel (int *a, int *b, int *c){
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < N){
        c[tid] = a[tid] + b[tid];
    }
}

int main(int argc, char *argv[]){
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;
    // allocate the memory on the GPU
    cudaMalloc( (void**)&dev_a, N * sizeof(int) );
    cudaMalloc( (void**)&dev_b, N * sizeof(int) );
    cudaMalloc( (void**)&dev_c, N * sizeof(int) );

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }
```

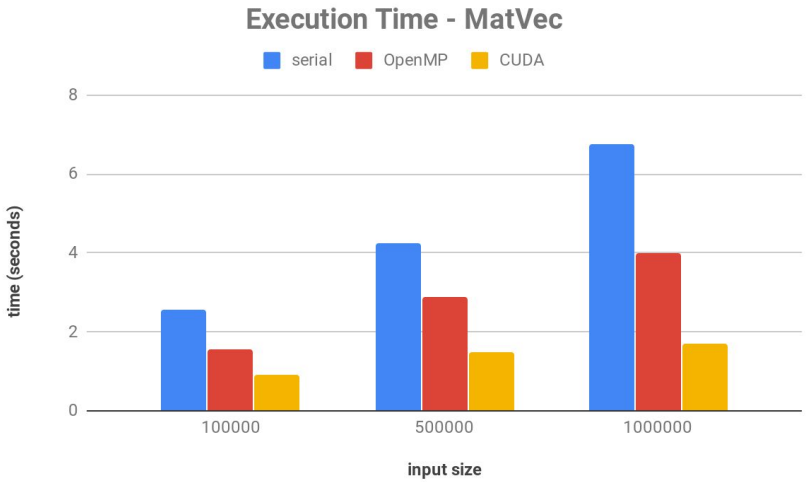
Modelo de Execução - Soma de Vetores

```
30
31      // copy the arrays 'a' and 'b' to the GPU
32      cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice );
33      cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice );
34      int threads_per_block = atoi(argv[1]);
35      dim3 dimBlock = threads_per_block;
36      dim3 dimGrid = ceil((int) N / threads_per_block);
37      kernel<<<dimGrid,dimBlock>>>( dev_a, dev_b, dev_c );
38
39      // copy the array 'c' back from the GPU to the CPU
40      cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost );
41
42      // display the results for
43      for (int i=0; i<N; i++){
44          printf( "%d + %d = %d\n", a[i], b[i], c[i] );
45      }
46      // free the memory allocated on the GPU
47      cudaFree( dev_a );
48      cudaFree( dev_b );
49      cudaFree( dev_c );
50      return 0;
51  }
```

Instruções para o TP III

Métricas de desempenho

input size	serial	OpenMP	CUDA
100000	2.56	1.55	0.9
500000	4.25	2.88	1.5
1000000	6.75	3.98	1.7



Leitura dos parâmetros:

```
int main(int argc, char ** argv){
  /*Usage*/
  if(argc < 2)
  {

    printf("Número inválido de argumentos");
    exit(-1);
  }

  /*parsing de command line args:*/
  /*formato:"./principal" opts: cpu gpu */
  if(!strcmp(argv[1], "CPU"))
  {
        mode = CPU;
        printf("MatVec em CPU \n");
  }
  else if(!strcmp(argv[1], "GPU"))
  {
        mode = GPU
              printf("MatVec em GPU \n");
  }
  ...
```

MatVec - Sequencial

```
/*Execucao sequencial em CPU*/
void MatrixVectorMult(double out[], double Val[], int RowPtr[], int Col[], double in[], int m){


   for (int i = 0; i < m; i++)
   {
      for (int k = RowPtr[i]; k < RowPtr[i+1]; k++)
      {

        out[i] += Val[k] * in[Col[k]];
      }
   }

}/*MatrixVectorMult*/
```