

# Super aceleração de algoritmos usando GPUs de última geração

**Dr. Mário Gazziro<sup>1</sup>**

mario.gazziro@ufabc.edu.br

Monitores: Daniel M. Lima<sup>2</sup> e Eduardo M. Real<sup>1</sup>

<sup>1</sup> Universidade Federal do ABC - UFABC/Santo André-SP

<sup>2</sup> Universidade de São Paulo - ICMC-USP/São Carlos-SP

[https://github.com/mariogazziro/AutoML\\_GPU](https://github.com/mariogazziro/AutoML_GPU)

15/05/2018

## Resumo

Curso prático de elaboração de algoritmos acelerados para GPUs com arquitetura Pascal – será utilizada Titan XP com 3500 núcleos, porém a maioria das técnicas abordadas se aplicam a todas as classes de GPU da fabricante NVIDIA. Após introdução sobre análise do paralelismo de algoritmos, serão apresentados estudos de caso aplicados em matrizes esparsas, ressonância magnética, e processamento de grafos em escalas de milhões. Em alguns dos casos apresentados a aceleração chega próximo de 30 vezes, e as técnicas apresentadas, como definição e ajuste fino da grade (GRID) de recursos, escolha dos tipos adequados de memória e técnicas para o escalonador de processos (WARP), podem ser facilmente adaptadas a outros problemas, logo os alunos inscritos são encorajados a trazer seus próprios problemas de pesquisa para paralelização em nossa oficina.

## CV resumido

Mário Gazziro possui graduação e doutorado pela USP, com especialização na Toshiba Semiconductor, Japão. Atualmente é professor na UFABC.

- GPGPU (ou simplesmente GPU): General-Purpose computing on Graphics Processing Units
- Speed-Up: Número adimensional que mede a performance relativa entre dois sistemas processando o mesmo problema.
- CUDA: Compute Unified Device Architecture
- TFlops - Teraflops: "Teraflop é uma unidade de poder de processamento, equivalente ao cálculo de um TRILHÃO de operações de ponto flutuante (IEEE 754) por segundo."

# Introdução

- GPU (*Graphics Processing Unit*) [1, 2] é uma arquitetura *a priori* derivada do paradigma DATAFLOW [3], pesquisado na década de 80.
  - ▶ Ultrapassou o mercado de jogos, indo muito além da computação gráfica, sendo amplamente utilizada em computação científica e number-crunching.
  - ▶ Possui características diferentes de CPU.
- Em comparação com CPU, o crescimento do número de núcleos é geométrico, embora sejam núcleos relativamente mais simples.
  - ▶ Causalidade e Paralelismo (nem tudo pode ser paralelizado)
  - ▶ Lei de *Amdhal* [4] (determinação do máximo *speed-up* teórico com múltiplos processadores).
- Supercomputação para as massas (série de artigos da revista Dr. Dobbs).
  - ▶ Se bem explorado, o poder computacional pode chegar a TFlops.
  - ▶ Porém, um ajuste adequado do problema e da sua paralelização são vitais para bons *speed-ups*.

# CPU x GPU

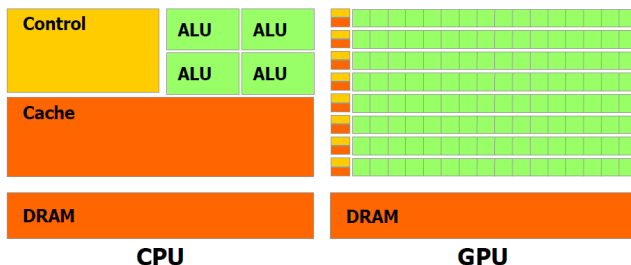


Figure: GPUs dedicam transistores para o processamento dos dados.[5]

- uma unidade de computação de GPU é muito mais simples que um moderno núcleo de CPU superescalar.
- uma unidade de computação de GPU não faz previsão de desvio.
- todos os núcleos da GPU executam as mesmas instruções, ao mesmo tempo, mas operam em dados diferentes (SIMD).
- um núcleo da CPU tem um cache grande, previsão de desvio e maior velocidade de clock.

- Uma arquitetura de computação paralela de propósito geral.
  - ▶ um modelo de programação paralela e arquitetura de conjunto de instruções que aproveita o mecanismo de computação paralela nas GPUs NVIDIA para resolver muitos problemas computacionais complexos de uma maneira mais eficiente do que em uma CPU.
- O CUDA vem com um ambiente de software que permite aos desenvolvedores usarem o C como uma linguagem de programação de alto nível.
  - ▶ Ideia de manter uma curva de aprendizado baixa para programadores familiarizados com linguagens de programação padrão, como C.

Núcleo com 3 abstrações principais (como um conjunto mínimo de extensões de linguagem)

- 1 Uma hierarquia de grupos de *threads*.
- 2 Memórias compartilhadas.
- 3 Sincronização de barreiras.

- Particionar o problema em sub-problemas que podem ser resolvidos independentemente em paralelo por blocos de *threads*.
- Cada sub-problema são pequenas partes que podem ser resolvidas cooperativamente em paralelo por todos os *threads* dentro do bloco.

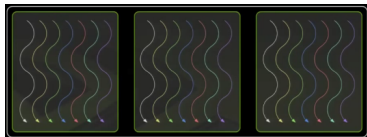
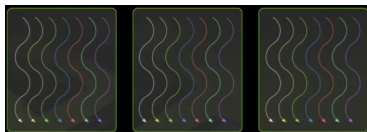
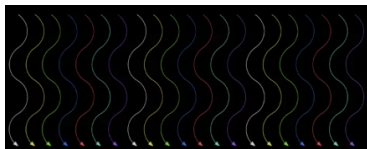
# Modelo de Programação

- host (CPU): executa a aplicação.
- device (GPU): executa **kernels**.
- host e device tem DRAMs (memórias) próprias.
- host:
  - ▶ Aloca memória no device.
  - ▶ Transfere dados de entrada para o device.
  - ▶ Dispara a execução de kernels.
  - ▶ Transfere dados resultantes do device para o host.
  - ▶ Libera memória no device.
- Kernel: executa no device N vezes em N threads em paralelo.



# Modelo de Programação

- Threads são organizadas em **blocos**.
- Um bloco é um arranjo 1D, 2D ou 3D (índices) de **threads**.
- Blocos são organizados em **grids**.
- Um grid é um arranjo 1D ou 2D (índices) de blocos.
  - ▶ Os blocos de um grid têm o mesmo número de threads.
- Hierarquia de threads:
  - ▶ Threads.
  - ▶ Bloco de threads (blocks).
  - ▶ Grade de blocos (grid).



# Modelo de Memória

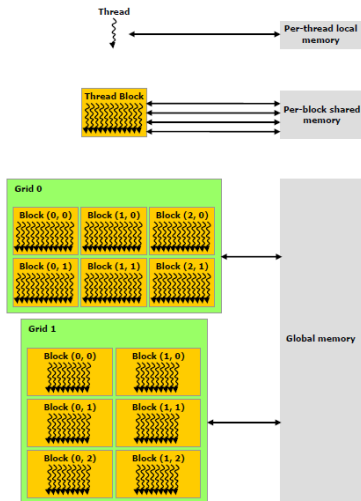


Figure: Hierarquia de Memória [5]

# Modelo de Programação

Grid de 2 dimensões com blocos de threads de 2 dimensões:

- 1 Grid 2D com:
  - ▶  $3 \times 2 \times 1 = 6$  blocos
- 2 Blocos 2D com:
  - ▶  $4 \times 3 \times 1 = 12$  threads por bloco.
- 3 Total de threads:
  - ▶  $6 \times 12 \times 1 = 72$  threads no total.

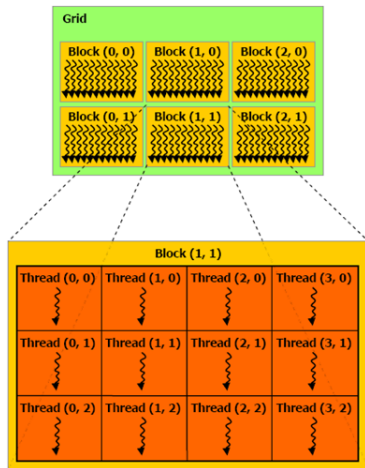
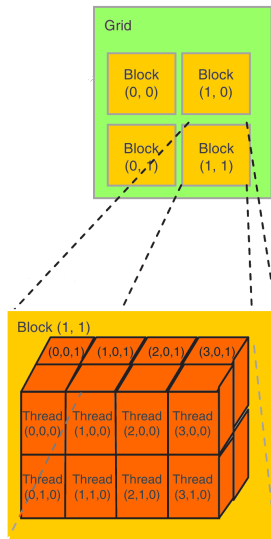


Figure: Grid (grade) de blocos de threads [5]

# Modelo de Programação

Grid de 2 dimensões com blocos de threads com 3 dimensões:

- 1 Grid 2D com:
  - ▶  $2 \times 2 \times 1 = 4$  blocos
- 2 Blocos 3D com:
  - ▶  $4 \times 2 \times 2 = 16$  threads por bloco.
- 3 Total de threads:
  - ▶  $4 \times 16 = 64$  threads no total.



## Identificação das threads

- Block
  - ▶ Thread ( $tx, ty, tz$ )
  - ▶ `threadIdx.dim`, i.e., (`threadIdx.x`, `threadIdx.y`, `threadIdx.z`)
- Grid
  - ▶ Block ( $bx, by, bz$ )
  - ▶ `blockIdx.dim`, i.e., (`blockIdx.x`, `blockIdx.y`, `blockIdx.z`)
- Gerações CUDA com diferentes números máximos de threads por bloco e de blocos por Grid.

# Modelo de Programação

## Identificação das threads

Especificação	Compute Capability					
	1.0	1.1	1.2	1.3	2.0	2.1
Número de dimensões da grade de blocos	2				3	
Tam. máx. de cada dimensão na grade	65535					
Número de dimensões do bloco de threads	3					
Tam. máx. das dimensões x e y no bloco	512				1024	
Tam. máx. da dimensão z no bloco	64					
Núm. máx. threads no bloco	512				1024	

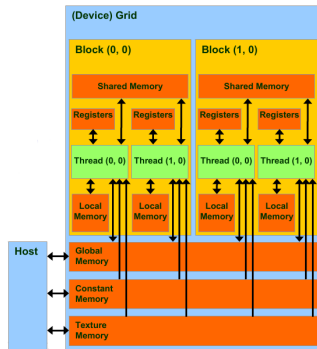
• e.g.

- ▶ Block, (threadIdx.x, threadIdx.y, threadIdx.z):
  - Tam. máx. cada dim: ( 512, 512, 64 ); máx. threads = 512.
  - Tam. máx. cada dim: ( 1024, 1024, 64 ); máx. threads = 1024.
- ▶ Grid, (blockIdx.x, blockIdx.y):
  - Tam. máx. cada dim: ( 65535, 65535 );
- ▶ Grid, (blockIdx.x, blockIdx.y, blockIdx.z):
  - Tam. máx. cada dim: ( 65535, 65535, 65535 );

# Modelo de Programação

Acesso da thread:

- compartilhada do bloco
  - ▶ Visível para todas e tempo de vida do bloco.
- local e registradores da thread.
- global, constante e leitura do Grid.

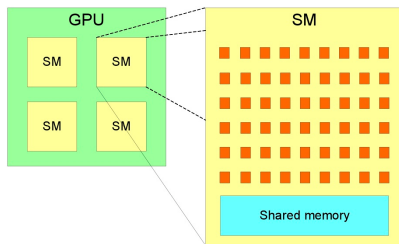


Tipo	Escopo	Acesso	Velocidade	Tempo de Vida
Registrador	Thread	R/W	Rápido	Kernel
Local	Thread	R/W	Lento	Kernel
Compartilhada	Bloco	R/W	Rápido	Kernel
Global	Grade	R/W	Lento	Aplicação
Constante	Grade	R/O	Rápido	Aplicação
Textura	Grade	R/O	Rápido	Aplicação

# Modelo de Programação

## SM:

- 128 cores CUDA formam 1 Streaming Processor (Titan XP).
- 30 Streaming Multiprocessor por GPU (Titan XP).
- Número máximo de WARPs deve ser menor ou igual ao número de SMs.
- As threads são atribuídas aos SMs em blocos.

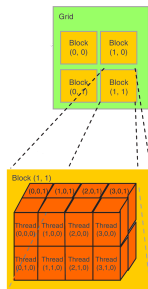
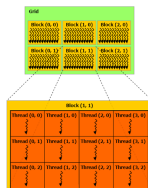




# Modelo de Programação

Mapeando o índice do thread no bloco:

- Threads podem ser identificados com índices de 1, 2 ou 3 dimensões (formando thread blocks de uma, duas ou três dimensões).
- Índice de uma thread:
  - ▶ Se for um bloco 1D: é a mesma coisa.
  - ▶ Se for um bloco 2D ( $D_x$ ,  $D_y$ ):  
threadId de um thread de índice  $(x, y)$  é  $x + yD_x$ .
  - ▶ Se for um bloco 3D ( $D_x$ ,  $D_y$ ,  $D_z$ ):  
threadId de uma thread de índice  $(x, y, z)$  é  $x + yD_x + zD_xD_y$ .

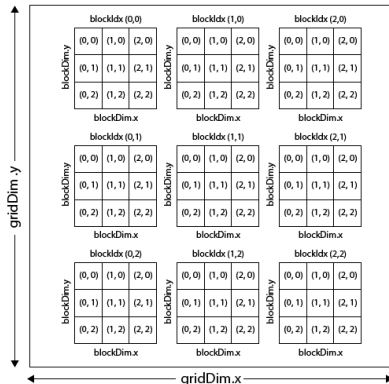


# Modelo de Programação

## CUDA Grid

Mapeando o índice do thread:

- # elementos por dimensão: *blockDim.x* e *blockDim.y*
- id bloco: *blockIdx.x*, *blockDim.y*
- Thread como uma matriz 2D: *thread[idx][idy]*


$$\begin{aligned} idx &= (blockIdx.x * blockDim.x) + threadIdx.x; \\ idy &= (blockIdx.y * blockDim.y) + threadIdx.y; \\ thread\_id &= (gridDim.x * blockDim.x * idy) + idx; \end{aligned}$$

## CUDA Thread Indexing Cheatsheet

<https://cs.calvin.edu/courses/cs/374/CUDA/CUDA-Thread-Indexing-Cheatsheet.pdf>

- Para um código ser executado na GPU, é necessário fazer parte de uma função que é chamada de **kernel**, que será executada pela GPU de forma paralela.

### Example (Kernel simples)

```
#include<stdio.h>
__global__ void mykernel (void){ //função kernel
printf("Oi GPU!\n");
}
int main(void){
    mykernel<<<1,1>>>(); //chamada para a função
    return 0;
}
```

## Example (Soma de vetores)

```
#include <stdio.h>

__global__ void somavet(float *a, float *b, float *c){
    int id = (blockDim.x * blockIdx.x) + threadIdx.x;
    c[id] = a[id] + b[threadIdx.x];
}

int main(void){
    ...
    somavet<<<nBlocos,nThreadsPorBloco>>>(gpu_a, gpu_b, gpu_c);
    ...
}
```

- Supondo um vetor de N elementos, solicitando **nBlocos** na chamada do kernel, com **nThreadsPorBloco** cada um.

## Example

```
int main(void){
    dim3 nThreadsPorBloco(3, 3);
    int nBlocos = 1;

    //invocando o kernel
    my_kernel<<<nBlocos,nThreadsPorBloco>>>(a, b, c);
    ...
}
```

- Os parâmetros são utilizados em tempo de execução para organizar as suas threads. O “nBlocos” define a dimensão do grid e o “nthreadsPorBloco” define a dimensão do bloco.
  - ▶ Cada um pode ser declarado como int ou dim3.
  - ▶ No exemplo, a dimensão dos blocos (“nthreadsPorBloco”) é definida como um arranjo de  $3 \times 3$  threads. A dimensão do grid (“numBlocos”) poderia ser definida de forma análoga, bem como utilizar uma terceira dimensão para qualquer um dos parâmetros.

## Example (Soma de vetores, com N elementos cada um)

```
__global__ void soma(float * a, float * b, float * c){
    int idx, idy, thread_idx;
    idx = blockIdx.x * blockDim.x + threadIdx.x;
    idy = blockIdx.y * blockDim.y + threadIdx.y;
    thread_idx = (gridDim.x * blockDim.x * idy) + idx;
    c[thread_idx] = a[thread_idx] + b[thread_idx];
}

int main(void){
    ...
    dim3 nThreadsPorBloco(threadx, thready);
    dim3 nBlocos(blocosx, blocosy);
    ...
    soma<<<nBlocos,nThreadsPorBloco>>>(gpu_a, gpu_b, gpu_c);
    ...
}
```

- $N = \text{blocosx} * \text{blocosy} * \text{threadx} * \text{thready}$

## Adição de matrizes elemento a elemento

### CPU Program

```
void add_matrix
( float* a, float* b, float* c, int N ) {
    int index;
    for ( int i = 0; i < N; ++i )
        for ( int j = 0; j < N; ++j ) {
            index = i + j*N;
            c[index] = a[index] + b[index];
        }
}

int main() {
    add_matrix( a, b, c, N );
}
```

### CUDA Program

```
--global__ add_matrix
( float* a, float* b, float* c, int N ) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N;
    if ( i < N && j < N )
        c[index] = a[index] + b[index];
}

int main() {
    dim3 dimBlock( blocksizes, blocksizes );
    dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
    add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N );
}
```



## Elementwise Matrix Addition

### CPU Program

```
void add_matrix  
( float* a, float* b, float* c, int N ) {  
    int index;  
    for ( int i = 0; i < N; ++i )  
        for ( int j = 0; j < N; ++j ) {  
            index = i + j*N;  
            c[index] = a[index] + b[index];  
        }  
}  
  
int main() {  
    add_matrix( a, b, c, N );  
}
```

### CUDA Program

```
--global__ add_matrix  
( float* a, float* b, float* c, int N ) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    int j = blockIdx.y * blockDim.y + threadIdx.y;  
    int index = i + j*N;  
    if ( i < N && j < N )  
        c[index] = a[index] + b[index];  
}  
  
int main() {  
    dim3 dimBlock( blockSize, blockSize );  
    dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );  
    add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N );  
}
```

- Programa básico em C para CUDA:

- ▶ Dispositivo a ser usado (`cudaSetDevice()`);
- ▶ Alocação de memória no host;
- ▶ Dados de entrada na memória do host;
- ▶ Alocação de memória no device (`cudaMalloc()`);
- ▶ Transferência de dados do host para device (`cudaMemcpy()`);
- ▶ Invocação do(s) kernel(s);
- ▶ Transferência de dados do device para host;
- ▶ Liberação de memória no host;
- ▶ Liberação de memória no device (`cudaFree()`).

## Exemplo:

```
#include <stdio.h>
#define threadx 8
#define thready 4
#define blocosx 2
#define blocosy 2
#define N      (blocosx*blocosy*threadx*thready)
__global__ void soma(float * a, float * b, float * c);
int main(void){
    unsigned int tamanho_bytes, i;
    float cpu_a[N], cpu_b[N], cpu_c[N], *gpu_a, *gpu_b, *gpu_c;
    dim3 nThreads(threadx, thready), nBlocos(blocosx, blocosy);

    //ler cpu_a e cpu_b

    /* alocando memoria na gpu */
    tamanho_bytes = N * sizeof(float);
    cudaMalloc((void **) &gpu_a, tamanho_bytes);
    cudaMalloc((void **) &gpu_b, tamanho_bytes);
    cudaMalloc((void **) &gpu_c, tamanho_bytes);

    /* Move os dados da CPU para GPU */
    cudaMemcpy(gpu_a, cpu_a, tamanho_bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(gpu_b, cpu_b, tamanho_bytes, cudaMemcpyHostToDevice);

    soma<<<nBlocos,nThreads>>>(gpu_a, gpu_b, gpu_c);

    /* Move os dados da CPU para GPU */
    cudaMemcpy(cpu_c, gpu_c, tamanho_bytes, cudaMemcpyDeviceToHost);

    //mostrar cpu_c

    cudaFree(gpu_a); cudaFree(gpu_b); cudaFree(gpu_c);
    return 0;
}
```

```
/* Kernel CUDA */
__global__ void soma(float * a, float * b, float * c){
    unsigned int idx, idy, thread_idx;

    /* Coordenas de um thread dentro do grid */
    idx      = blockIdx.x * blockDim.x      + threadIdx.x;
    idy      = blockIdx.y * blockDim.y      + threadIdx.y;

    /* identificador de um thread com coordenasa idx e idy */
    thread_idx = (gridDim.x * blockDim.x * idy) + idx;
    c[thread_idx] = a[thread_idx] + b[thread_idx];
}
```

```

Device 0: "TITAN Xp"
  CUDA Driver Version / Runtime Version      9.1 / 9.1
  CUDA Capability Major/Minor version number: 6.1
  Total amount of global memory:             12196 MBytes (12788498432
bytes)
  (30) Multiprocessors, (128) CUDA Cores/MP: 3840 CUDA Cores
  GPU Max Clock rate:                        1582 MHz (1.58 GHz)
  Memory Clock rate:                          5705 Mhz
  Memory Bus Width:                           384-bit
  L2 Cache Size:                             3145728 bytes
  Maximum Texture Dimension Size (x,y,z)     1D=(131072), 2D=(131072,
65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers 1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers 2D=(32768, 32768), 2048 layers
  Total amount of constant memory:            65536 bytes
  Total amount of shared memory per block:    49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                 32
  Maximum number of threads per multiprocessor: 2048
  Maximum number of threads per block:        1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                      2147483647 bytes
  Texture alignment:                         512 bytes
  Concurrent copy and kernel execution:       Yes with 2 copy engine(s)
  Run time limit on kernels:                  No
  Integrated GPU sharing Host Memory:         No
  Support host page-locked memory mapping:    Yes
  Alignment requirement for Surfaces:         Yes
  Device has ECC support:                     Disabled
  Device supports Unified Addressing (UVA):   Yes
  Supports Cooperative Kernel Launch:         Yes
  Supports MultiDevice Co-op Kernel Launch:   Yes
  Device PCI Domain ID / Bus ID / location ID: 0 / 1 / 0
  Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device
simultaneously) >

```

# Exemplos

*Apresentação dos estudos de caso. Códigos disponíveis em:*  
*[https://github.com/mariogazziro/AutoML\\_GPU](https://github.com/mariogazziro/AutoML_GPU)*

# Estudo de caso 1: Operações sobre matrizes esparsas

```
N=32000;
K(1:N) = sparse(rand(1,N));
g1(1:2*N) = sparse(rand(1,2*N));
k = 1.3;
for i=1:N
    for j=1:N
        M(i,j)=g1(N+i-j)*(K(i)+k)*(K(j)+k);
    end;
end;
```

[https://github.com/mariogazziro/AutoML\\_GPU/tree/master/sparse\\_mat](https://github.com/mariogazziro/AutoML_GPU/tree/master/sparse_mat)

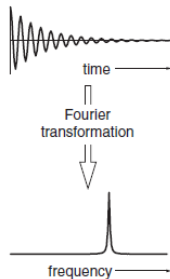
# Estudo de caso 1: Operações sobre matrizes esparsas

N	CPU(seg)		GPU(seg)	SPEED-UP
1000	0.002521	/	0.126500	0.019928853754940
2000	0.009467	/	0.122049	0.077567206613737
4000	0.065304	/	0.139727	0.467368511454479
8000	0.449128	/	0.215426	2.084836556404519
16000	2.834458	/	0.520717	5.443375192282948
32000	11.846775	/	1.701548	6.962351341249262

Alterar o tamanho do bloco de 128 para 1024 (máximo Titan XP) e verificar novamente os resultados.

## Estudo de caso 2: Ajuste de fase em RMN

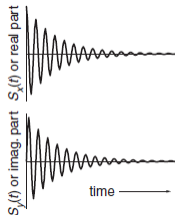
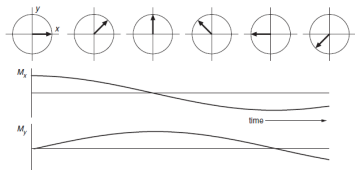
- O *free induction signal* (em que a precessão acontece na frequência de Larmor) sofre uma ação de relaxamento, sendo chamado de *free induction decay* ou **FID**, um signal no domínio do tempo.
- **A ideia é transformar este sinal, que depende do tempo, no espectro, no qual o eixo horizontal é a frequência.**
  - ▶ Esses FIDs podem ser somados e agregados, isso se converte em signal de domínio de frequência por uma transformada de Fourier.
- A transformação de Fourier é o processo matemático que nos leva de uma função do tempo (o domínio do tempo) - como um FID - a uma função no domínio da frequência - o espectro.





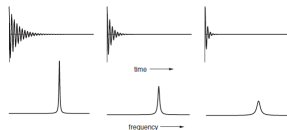
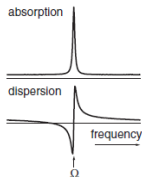
## Estudo de caso 2: Ajuste de fase em RMN

- O FID é um sinal complexo, com as partes real e imaginária correspondentes aos componentes x e y do sinal.
  - ▶ A magnetização transversal decai ao longo do tempo.



## Estudo de caso 2: Ajuste de fase em RMN

- Picos (Lorentzianos) formam a porção absorptiva, real, sendo que a parte imaginária é representada pela porção dispersiva do sinal. A área da parte absorptiva deve tender ao máximo, já a área da parte absorptiva deve tender a zero, com o cancelamento das partes positivas e negativas.
- Quanto mais rapidamente o FID decai, mais larga a linha no espectro correspondente.



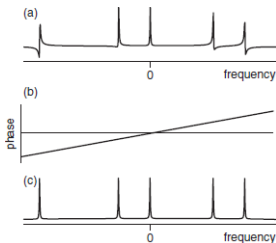
- Ao integrar as linhas no espectro, podemos determinar o número relativo de prótons (tipicamente) que contribuem para cada um deles.

## Estudo de caso 2: Ajuste de fase em RMN

- No entanto, o sinal pode estar deslocado em fase, apresentando um erro de fase.
  - ▶ A aparência tradicional do espectro depende da posição do sinal no tempo zero, i.e., na fase do sinal no tempo zero.
  - ▶ Poder haver uma mudança de fase desconhecida, levando a uma situação na qual não é mostrada uma linha de absorção clara na parte real, o que dificulta ou mesmo impossibilita a análise espectral.
  - ▶ Uma "correção" de fase pode ser aplicada ao espectro, usando diferentes modos.

## Estudo de caso 2: Ajuste de fase em RMN

- Ajustes de fase podem ser realizados em 2 ordens:
  - ▶ Ordem 0, onde apenas componentes do sinal são "transferidos" da parte imaginária para a parte real, por rotação complexa;
  - ▶ Ordem 1, onde, somado a rotação complexa, é acrescentado um fator de peso com base em um pivô e um ângulo de rotação.
- Ilustração de ajuste de fase de ordem 1, com pivô no centro do espectro:



- O ajuste de fase demanda variações de fatores de ordem 0, 1 e pivô.
  - ▶ Ou seja, é possível variar fatores de ordem 0 (entre 0 e 360 graus), fatores de ordem 1 (entre 0 e 360 graus), assim como a origem do pivô (que pode ser em qualquer ponto do espectro).

## Estudo de caso 2: Ajuste de fase em RMN

Minimização de Entropia:

$$\text{Min}E = \sum_i h_i \ln h_i + P(R_i) \qquad h_i = \frac{|R_i|}{\sum_i |R_i|}$$

$$R_i = R_i \cos(\Phi_i) - I_i \sin(\Phi_i) \qquad \Phi_i = \text{phc0} + \text{phc1} \times \frac{i}{n}$$

- $\text{Min}E \rightarrow$  função de minimização (mínimos quadrados).
- $h_i \rightarrow$  derivadas normalizadas do espectro (FFT) da RMN.
- $P \rightarrow$  função de penalidade para evitar bandas negativas.

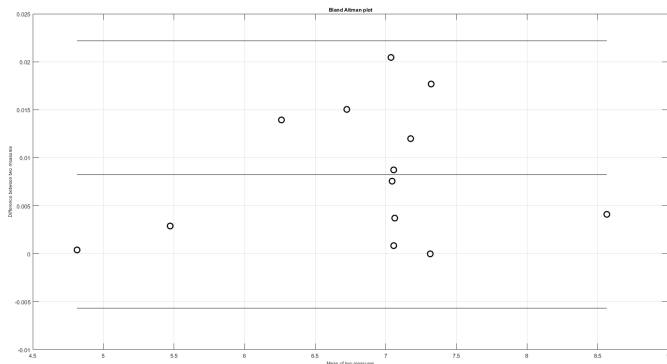
## Estudo de caso 2: Ajuste de fase em RMN

https:

`//github.com/mariogazziro/AutoML_GPU/tree/master/phase_adj`

## Estudo de caso 2: Ajuste de fase em RMN

Gráfico comparativo (Bland-Altman) entre os resultados das entropias calculadas pela CPU e GPU



## Estudo de caso 3: Processamento de grafos em escala de bilhões - Demonstração prática entre Gunrock (GPU) vs M-Flash (CPU)

<https://github.com/gunrock/gunrock>

<http://images.nvidia.com/events/sc15/pdfs/>

[SC5139-gunrock-multi-gpu-processing-library.pdf](#)

<https://www.groundai.com/project/>

[m-flash-fast-billion-scale-graph-computation-using-a-bimodal-](#)

<https://github.com/M-Flash>



# Estudo de caso 3: Processamento de grafos em escala de bilhões - Demonstração prática entre Gunrock (GPU) vs M-Flash (CPU)

base: com-orkut.ungraph.txt

tamanho 3,07M vértices com 117,19M arestas

tamanho em disco: 1.65GB

tempo do gunrock (GPU) ? — tempo do mflash (CPU) ?

# Estudo de caso 3: Processamento de grafos em escala de bilhões - Demonstração prática entre Gunrock (GPU) vs M-Flash (CPU)

base: com-orkut.ungraph.txt

tamanho 3,07M vértices com 117,19M arestas

tamanho em disco: 1.65GB

tempo do gunrock (GPU) 405.98s — tempo do mflash (CPU) ?

# Estudo de caso 3: Processamento de grafos em escala de bilhões - Demonstração prática entre Gunrock (GPU) vs M-Flash (CPU)

base: com-orkut.ungraph.txt

tamanho 3,07M vértices com 117,19M arestas

tamanho em disco: 1.65GB

tempo do gunrock (GPU) 405.98s — tempo do mflash (CPU) 54.69s

mflash 7,42x mais rápido que que gunrock!

# Ocupância

## - Definição:

Ocupância é a razão entre o número de multiprocessadores utilizados e o total disponível (não necessita ser um número inteiro, visto que recursos - registradores ou memória compartilhada - de um multiprocessador podem ser utilizados parcialmente).

## - Cálculo da ocupância:

Usar a calculadora online (que segue cálculos definidos por um white-paper da nvidia):

<https://xmartlabs.github.io/cuda-calculator/>

com os seguintes parâmetros para determinar a ocupância do software de ajuste de fase em RMN:

Compute Capability version: 6.1

Threads per block: 360

Registers per thread: 32

Shared memory per block: 49152

# Ocupância

Explicando de onde vem cada numero:

A capacidade de computacao da Titan XP 6.1, número que pode ser verificado com o aplicativo devicequery do cuda.

Já "threads per block" foi definido como 360, pois o problema do estudo de caso lança os blocos com as dimensões (360,1,1), visto que ele faz rotacoes complexas entre  $-\pi$  e  $+\pi$ .. em suma, é um parâmetro do problema em questão.

"Registradores por thread" e "shared memory per block" foram retirados do log gerado pelo compilador nvcc, usando um flag extra na compilação do kernel (parte em C):






# Ocupância

PTXAS Info sobre alocação do brute\_force\_phase\_adjust.py

```
ptxas info      : 8 bytes gmem, 49156 bytes cmem[3]
ptxas info      : Compiling entry function 'phase_adj' for 'sm_60'
ptxas info      : Function properties for phase_adj
                  98304 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info      : Used 32 registers, 49152 bytes smem, 360 bytes gmem
```

A ocupância calculada foi de 37,5%.

# Referências

-  NVIDIA Corporation. Graphics Processing Unit (GPU). Disponível em: <http://www.nvidia.com/object/gpu.html>. Acesso em: mai-2018.
-  NVIDIA Corporation. CUDA GPUs. Disponível em: <https://developer.nvidia.com/cuda-gpus>. Acesso em: mai-2018.
-  SOUSA, T. B. Dataflow Programming: Concept, Languages and Applications. In. Doctoral Symposium on Informatics Engineering. 2012.
-  RODGERS, D. P. Improvements in multiprocessor system design. In. Proceedings of the 12th annual international symposium on Computer architecture. Vol.13, 1985, p.225-231.
-  NVIDIA CUDA. NVIDIA CUDA C Programming Guide. Version 4.2, 2012.

# Roteiro da oficina GPU

- após ligar seu computador, selecionar a opção GNU Linux;
- usar como login: usuarios\\1242854  
(ou usuarios\\##### seu numero usp)
- usar senha: swatch\_X66642  
(ou no caso do seu número usp, usar sua senha do jupiter)
- acessar o aplicativo:  
Acessorios -> LXterminal



## Roteiro da oficina GPU - continuação

- receba um número (em papel) de 1 a 50 de um dos monitores;
- digite o comando abaixo, usando seu numero no lugar de XX:  

```
$ssh -Y uXX@10.11.16.171 -oProxyCommand=\`ssh gazziro@143.107.183.147 nc %h %p`
```
- digite 'yes' seguido de enter.
- use a senha (servidor intermediário): swatch\_X66642
- digite 'yes' seguido de enter.
- digite a segunda senha (servidor com a GPU): milhoverde
- testar o redirecionamento do servidor gráfico X:  

```
$xeyes &
```

# Roteiro da oficina GPU - Exemplos apresentados na parte teórica

- descomprima e rode o exemplo de matrizes esparsas:

```
$unzip sparse_mat.zip
```

```
$cd sparse_mat
```

```
$make
```

- notar que o comando 'make' já executa o script 'run', apresentando o resultado de tempo e speed-up para diversos tamanhos de matrizes em CPU e GPU;

# Roteiro da oficina GPU - Exemplos apresentados na parte teórica

```
$cd ~
```

-descomprima e rode o exemplo de ajuste de fase:

```
$unzip phase_adj.zip
```

```
$cd phase_adj
```

```
$python phase_adj.py Carlos10_figado
```

- verifique a parte absorptiva (direita) e dispersiva (esquerda) nos graficos de CPU (acima) e GPU (abaixo)

- plote gráfico comparativo bland-altman (hard-coded data):

```
$python bland-altman.py
```

# Roteiro da oficina GPU - Exercícios

```
$cd ~
```

```
$unzip intro.zip
```

```
$cd intro
```

- Edite o arquivo `intro.cu` e complete o código nas partes indicadas:

```
$edit intro.cu
```

- compile e teste com:

```
$nvcc -o intro intro.cu
```

```
$./intro
```

- após 15 minutos a solução será copiada para todas as pastas com o nome `intro-solution.zip`

## Roteiro da oficina GPU - Exercícios

```
$cd ~  
$unzip reconstruct.zip  
$cd reconstruct  
$make  
$./reconstruct  
$mv output300.jpg output300_100.jpg  
$eog output300_100.jpg &  
- Anote os tempos de CPU e GPU para 100 iterações (default)  
$edit reconstruct.h  
#DEFINE ITERATIONS 1000  
$make  
$./reconstruct  
$mv output300.jpg output300_1000.jpg  
$eog output300_1000.jpg &  
- Anote os tempos de CPU e GPU para 1000 iterações  
- Calcule os speed_up's da GPU sobre a CPU em ambos os casos.
```

# Roteiro da oficina GPU - Exercícios

- Redimensione o grid do problema anterior de 256 threads por bloco para um grid bidimensional de 16x16 threads por bloco;
- Não esquecer de ajustar os índices no kernel adequadamente;
- Para isso, será necessário editar 3 arquivos:  
`reconstruct.h`, `reconstruct.cu` e `reconstruct_kernels.cu`
- Execute com o novo grid e compare o speed-up com relação ao exemplo unidimensional (256 threads) de 1000 iterações.