Aspectos de otimização no desenvolvimento de aplicações em CUDA

Denise Stringhini (Mackenzie) - Alfredo Goldman (IME/USP)

II Escola Regional de Alto Desempenho de São Paulo

28/07/2011

Sumário

Introdução
Tópicos

- Computação heterogênea
- NVIDIA: Arquitetura Fermi
  - Visão geral
  - *Streaming multiprocessor* (SM)
  - Hierarquia de memória cache

Computação heterogênea

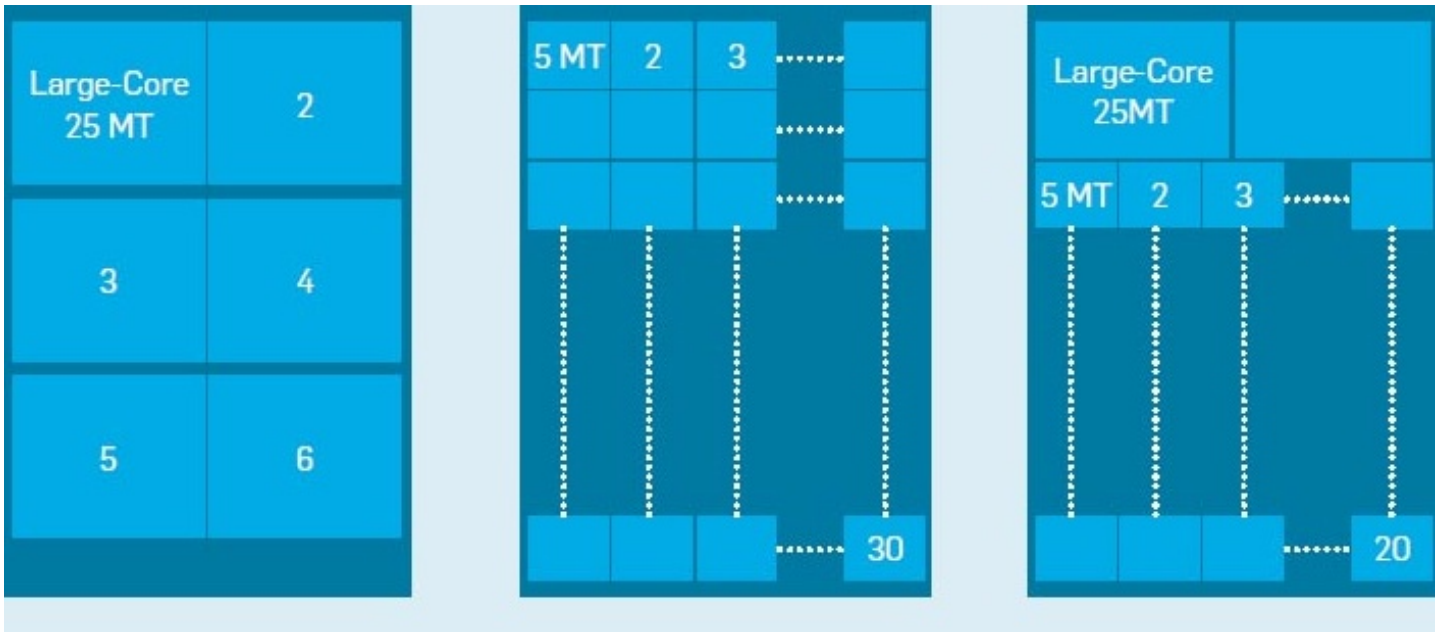
- Desempenho nos últimos 30 anos:
  - velocidade de clock
  - otimização das instruções
  - níveis de cache
  - *obtenção de desempenho: compilação*

Computação heterogênea

- Desempenho nos últimos 30 anos:
  - velocidade de clock
  - otimização das instruções
  - níveis de cache
  - *obtenção de desempenho: compilação*
- Próximos 20 anos:
  - gerenciamento de energia
  - otimização da movimentação de dados
  - **computação heterogênea**
  - *obtenção de desempenho: paralelização do código*
    - uso de bibliotecas, ferramentas de auto-tuning, etc

Computação heterogênea

- Projeção da Lei de Moore para 20 anos: 150 milhões de transístores.
  - Cenários para a organização lógica de 150 milhões de transístores em vários cores:

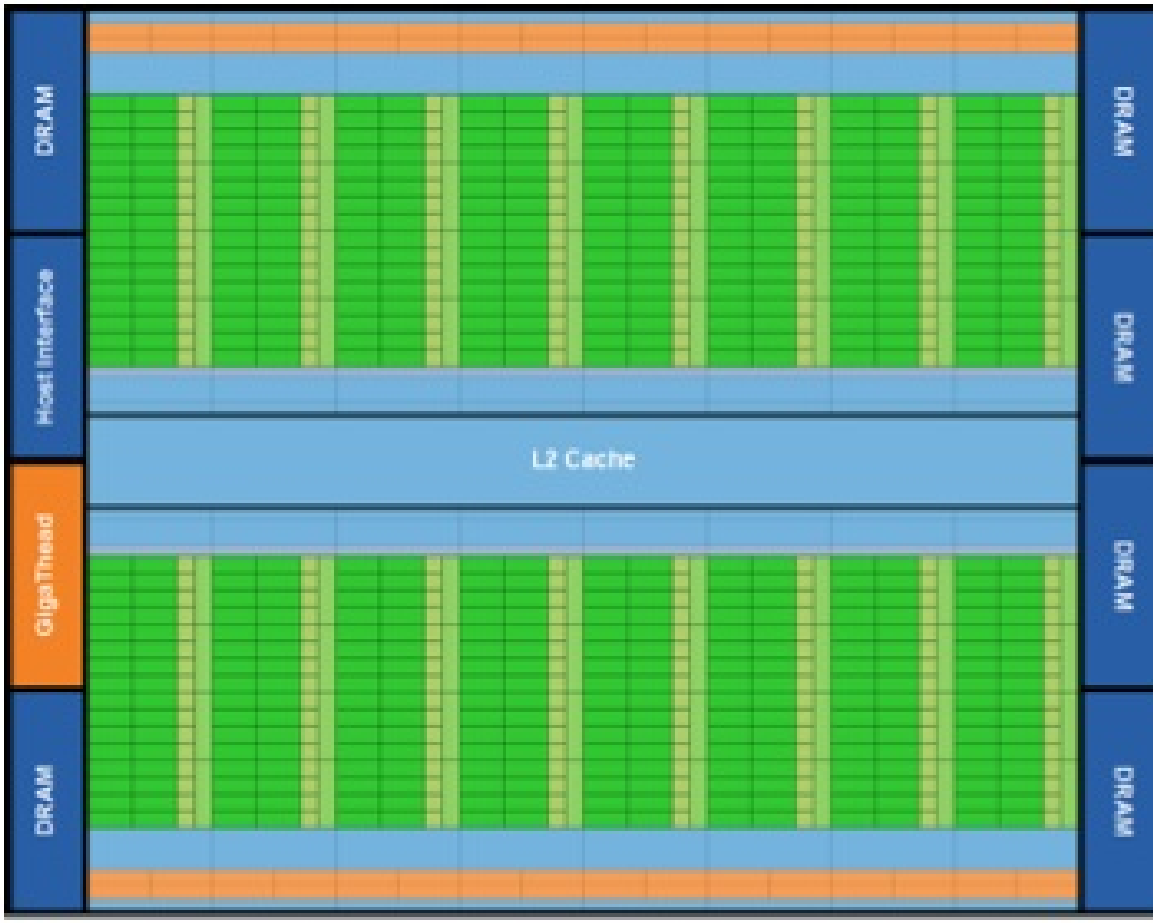


Multicore x manycore

- Multicore (CPU): 4 cores independentes, hyperthreading, instruções x86, despacho múltiplo e fora-de-ordem (ex: Intel Core i7).
- Manycore (GPU): 512 cores, compartilhamento do controle e cache L1, despacho único e em ordem (ex: GTX 580).
- O modelo de GPU dedica mais transístores para unidades de execução.

NVIDIA: Arquitetura Fermi
Visão geral

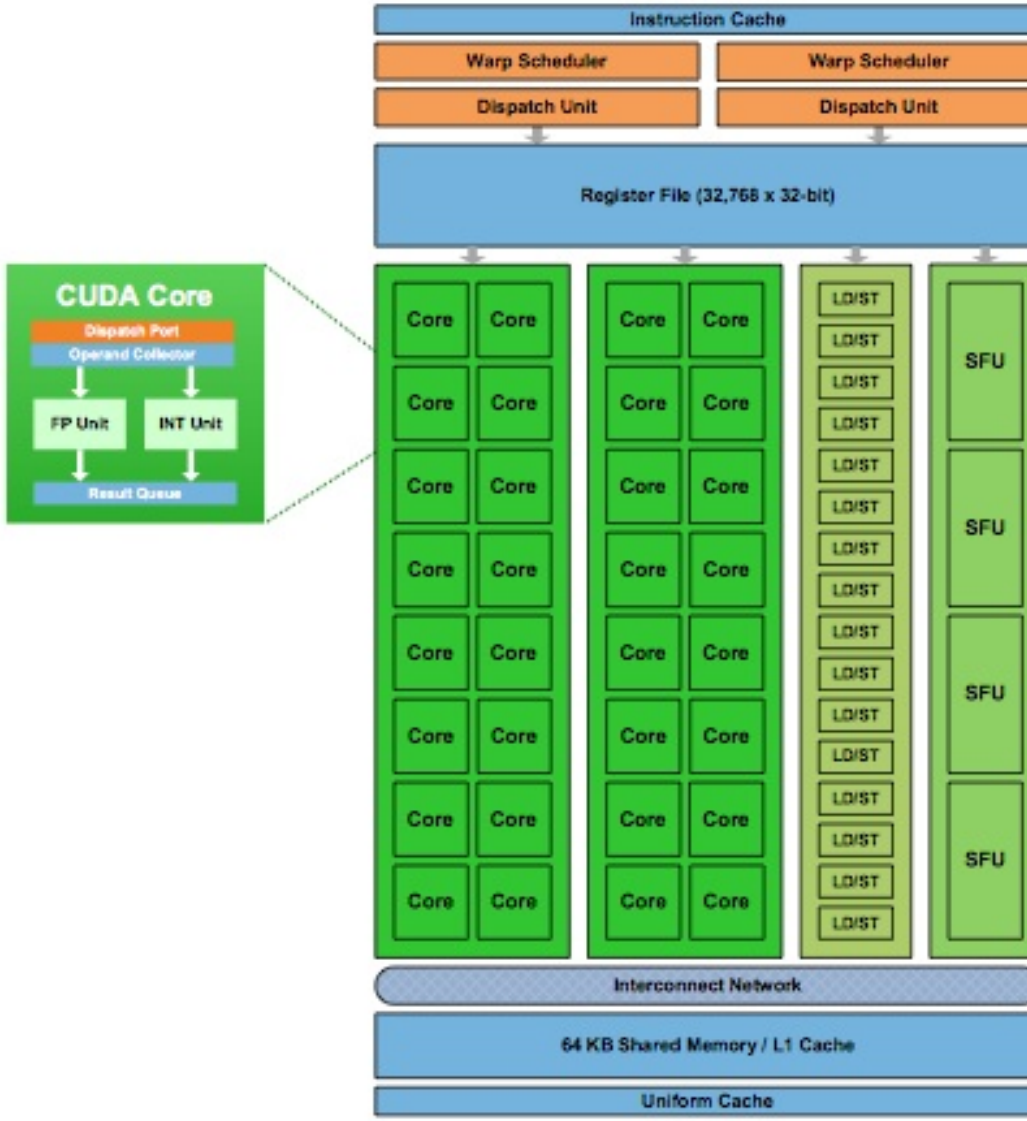
- 16 SMs (Streaming Multiprocessors)
  - 32 cores em cada SM
  - 512 cores no total
- Dois níveis de cache
- GigaThread *engine*
  - gerencia milhares de threads
  - permite a execução de kernels concorrentes



NVIDIA: Arquitetura Fermi

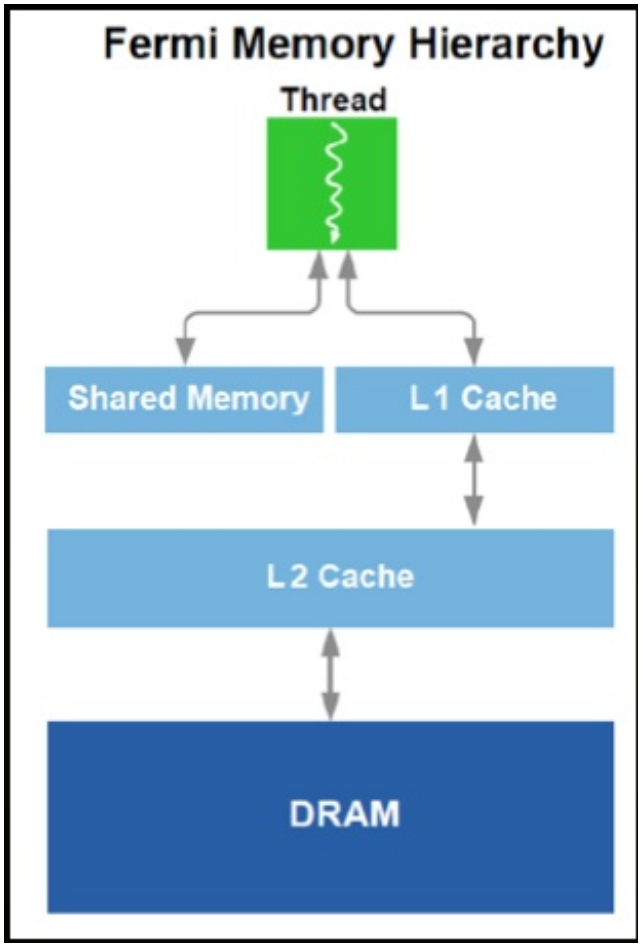
Streaming Multiprocessor (SM)

- 2 *warp schedulers* para 4 blocos de execução
  - 2 blocos de 16 cores
  - 1 bloco com 16 unidades load/store
  - 1 bloco com 4 unidades SFU
- Cache L1 / *shared memory*
- 32KB de registradores
- SFUs: execução de instruções especiais (sin, cos...)



NVIDIA: Arquitetura Fermi
Hierarquia de memória cache

- Primeira GPU a implementar *cache* na hierarquia de memória - combinada com a *shared memory*.
- Cache L1: 64KB
  - 16KB cache - 48KB shmem
  - 48KB cache - 16KB shmem (*default*)
- Cache L2: 768 KB
  - compartilhada entre todas as threads
- Memória global (DRAM): até 6GB

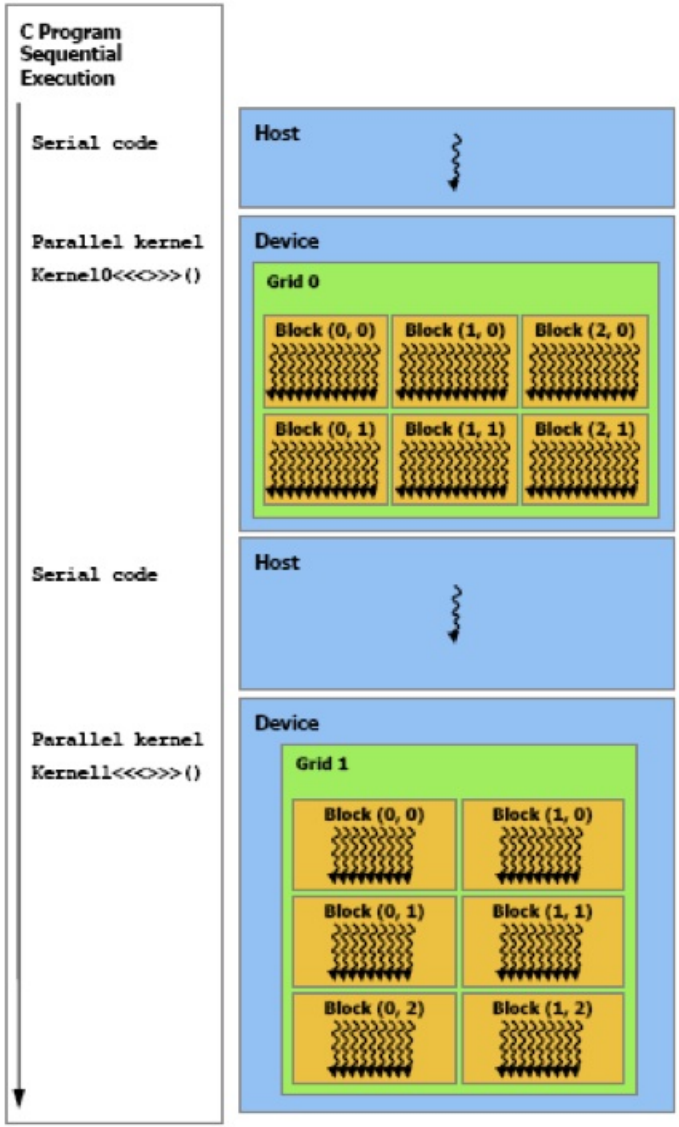


CUDA - Revisão
Tópicos

- Modelo de programação
- Modelo de memória
- Organização das threads
- Gerenciamento de memória
- Lançamento de kernel

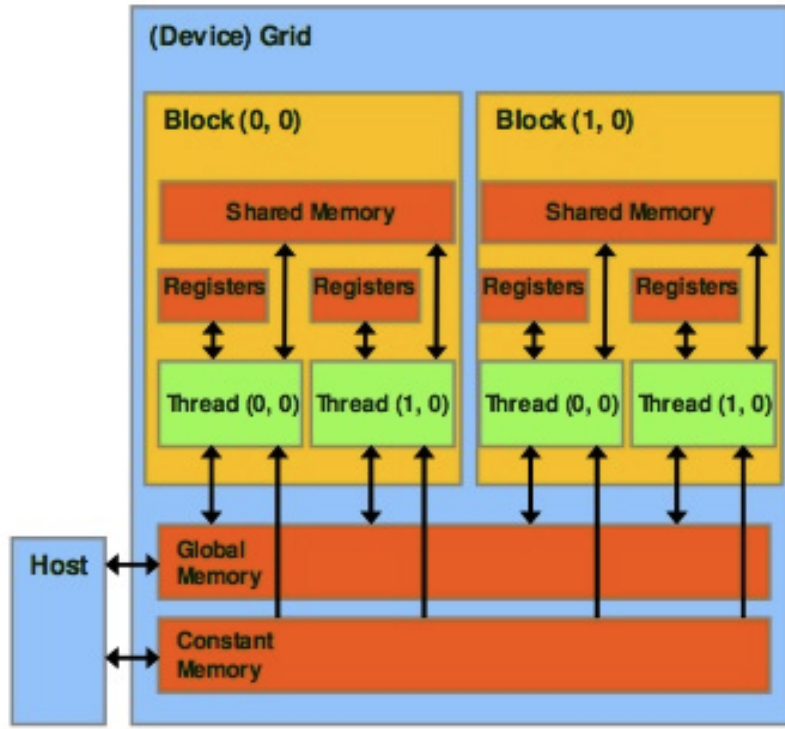
Modelo de programação

- Computação heterogênea: *host* e *device*.
- Código C estendido: função do tipo *kernel* executa no device.
  - O **nvcc** separa os dois durante a fase de compilação.
  - O device atua como um co-processador que executa em paralelo com o *host*.
  - Dados devem ser transferidos para a memória do *device*.
- Kernels são organizados em *grids* (até 2D) de vários blocos de threads (até 3D).
- CUDA 4.0 facilita o uso de várias threads no *host* e vários kernels no *device*.

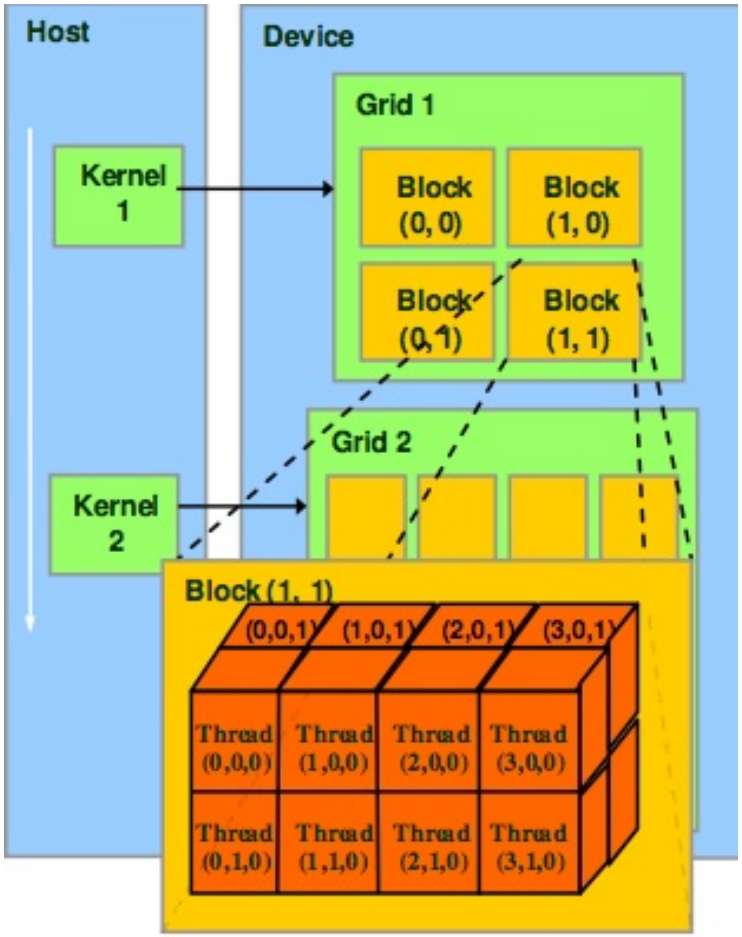


Modelo de memória

- Registradores:
  - Locais a cada thread.
  - Quantidade por thread é definida pelo compilador.
- Memória compartilhada (*shared*):
  - Compartilhada por threads no mesmo bloco.
  - Reduz a latência no acesso à memória global.
  - Alocada no código do kernel (__shared__).
  - Fermi: dividida com a cache.
- Memória global:
  - Acessada por todas as threads num mesmo *grid*.
  - Gerenciada a partir do *host*.
  - Fermi: dados mapeados para cache L2.



Organização das threads




```
dim3 dimBlock(4, 2, 2);
dim3 dimGrid(2, 2, 1);
KernelFunction<<<dimGrid, dimBlock>>>(...);
```

Gerenciamento de memória

- cudaMalloc(), cudaFree()
- cudaMemcpy()

```
// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

Lançamento de kernel

- Configuração do kernel:
  - número de blocos (int ou dim3)
  - threads por bloco (int ou dim3)

```
    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

Declaração de funções

- Modificadores definem onde a função será executada e de onde pode ser invocada:
  - __global__ (device/host)
  - __device__ (device/device)
  - __host__ (host/host)

```
__global__ void VecAdd(float* A, float* B, float* C, int N)
```

Variáveis pré-definidas

- Permitem que cada thread possa se distinguir entre as demais, permitndo que possam determinar sua área de trabalho.
  - **gridDim.x, gridDim.y**: quantidade de blocos em cada dimensão do *grid*
  - **blockDim.x, blockDim.y, blockDim.z**: quantidade de threads em cada dimensão do bloco
  - **blockId.x, blockId.y**: posição do bloco no *grid*
  - **threadId.x, threadId.y, threadId.z**: posicão da thread no bloco

```
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

Medidas de desempenho
Tópicos

- *Speedup*
- GFlops
- *Bandwidth*
- *Wall time*
  - Temporização na CPU
  - Temporização na GPU

Possibilidades
Discussão

- O que você faria com apenas 1 nó ?
- O que você faria com 100 nós ?
- O que você faria com 10.000 nós ?

Possibilidades
Discussão

- O que você faria com apenas 1 nó ?
- O que você faria com 100 nós ?
- O que você faria com 10.000 nós ?

Resposta

Depende do problema !

Speedup
Definição

- Motivação: Saber o quanto mais rápido é o algoritmo paralelo
- Com mais processadores, espera-se mais desempenho
- Medida simples: *speedup* $S_p = \frac{T_1}{T_p}$, onde
  - S_p *speedup* com p nós;
  - T_1 tempo de processamento com 1 nó;
  - T_p tempo de processamento com p nós.

Speedup
Mas, nem tudo é tão simples

- Alguns autores se enganam com respeito a esta medida
- Principal erro, se basear em um programa paralelo para a medida T_1
  O certo é usar o melhor sequencial !
- Existe *speedup* super linear ?

Speedup
Exemplo simples de como obter *speedup* superlinear

- Problema: Contar ocorrências de um número em um vetor grande desordenado
- Solução sequencial: Percorrer o vetor contando as ocorrências
- Solução paralela

Speedup
Limite de *speedup*: Amdahl's Law

- Se queremos obter um *speedup* de 80 com 100 processadores, qual razão do programa original pode ser sequencial ?
- Amdahl's Law $S_p = \frac{1}{\frac{P}{p}+(1-P)}$

Speedup
Limite de *speedup*: Amdahl's Law

- Se queremos obter um *speedup* de 80 com 100 processadores, qual razão do programa original pode ser sequencial ?
- Amdahl's Law $S_p = \frac{1}{\frac{P}{p}+(1-P)}$

  $80 = \frac{1}{\frac{P}{100}+(1-P)}$

Speedup
Limite de *speedup*: Amdahl's Law

- Se queremos obter um *speedup* de 80 com 100 processadores, qual razão do programa original pode ser sequencial ?
- Amdahl's Law $S_p = \frac{1}{\frac{P}{p}+(1-P)}$

  $80 = \frac{1}{\frac{P}{100}+(1-P)}$

  $frac_{paralela} = 0.99747...$
- Somente 0.25% do programa pode ser sequencial

Speedup
Mensagem importante

- No final, o grau de paralelismo depende muito do problema
- Ao menos existem muitos problemas trivialmente paralelizáveis !

GFlops
Definição

- Precisamos de uma forma de se medir o desempenho
- Forma simples, medir o número de instruções por unidade de tempo
- No caso, instruções de ponto flutuante

GFlops
Definição

- Precisamos de uma forma de se medir o desempenho
- Forma simples, medir o número de instruções por unidade de tempo
- No caso, instruções de ponto flutuante

- Primeira armadilha: complexidade das instruções
- Soma, subtração e multiplicação são fáceis de paralelizar
- Divisão não

GFlops
Continuação

- Segunda armadilha

GFlops
Continuação

- Segunda armadilha
- Precisão simples/Precisão dupla

GFlops
Continuação

- Segunda armadilha
- Precisão simples/Precisão dupla

- Mas, mesmo assim é uma medida aceita
- e no caso paralelo ?

GFlops
Continuação

- Segunda armadilha
- Precisão simples/Precisão dupla

- Mas, mesmo assim é uma medida aceita
- e no caso paralelo ?
- Soma das capacidades individuais fornece desempenho de pico
- Existem vários benchmarks para paralelismo

GFlops
Benchmark

- Motivação: Precisamos comparar máquinas/ambientes

GFlops
Benchmark

- Motivação: Precisamos comparar máquinas/ambientes
- Ideia: Criar um conjunto **representativo** de programas
- Armadilha: o significado de representativo é subjetivo
- Mas, existem medidas bem aceitas
- Linpack (sistemas densos de equações lineares)

GFlops
Na lista top500, temos apenas PFlops :)

- Lista de junho de 2011

Rank	Site	Computer
1	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu
2	National Supercomputing Center in Tianjin China	Tianhe-1A - NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C NUDT
3	DOE/SC/Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron 6-core 2.6 GHz Cray Inc.
4	National Supercomputing Centre in Shenzhen (NSCS) China	Nebulae - Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU Dawning
5	GSIC Center, Tokyo Institute of Technology Japan	TSUBAME 2.0 - HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows NEC/HP
6	DOE/NNSA/LANL/SNL United States	Cielo - Cray XE6 8-core 2.4 GHz Cray Inc.
7	NASA/Ames Research Center/NAS United States	Pleiades - SGI Altix ICE 8200EX/8400EX, Xeon HT QC 3.0/Xeon 5570/5670 2.93 Ghz, Infiniband SGI
8	DOE/SC/LBNL/NERSC United States	Hopper - Cray XE6 12-core 2.1 GHz Cray Inc.
9	Commissariat a l'Energie Atomique (CEA) France	Tera-100 - Bull bullx super-node S6010/S6030 Bull SA
10	DOE/NNSA/LANL United States	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband IBM

GFlops
Voltando aos benchmarks

- Em ambientes paralelos o custo de comunicação é importante
- Existem outros benchmarks como: SPEC HPG

- Existem benchmarks para GPUs
- Lembrete: benchmarks representam tipos de programas

Bandwidth
Definição

- Banda passante, vazão
- Medida de quando se passa entre dois pontos
- Muito importante em GPUs
- Não exatamente internamente, mas sim entre a CPU/Memória e a GPU
- Este tempo de transferência também tem que ser considerado

Wall time

- *Wall time* ou *elapsed time*: tempo total de execução de um determinado trecho de código.
- CPU: o SO oferece uma série de *timers* que podem ser usados em código.

```
//esquema para o cálculo de walltime
double zero, t0, t1, timer;
t0 = timer(t0);
...
//código a ser medido
...
t1 = timer(t0); //t1 contém o tempo em segundos
```

Temporização na CPU

Exemplo: gettimeofday()

- Vantagem: precisão em segundos e microsegundos
- Exemplo em linguagem C com conversão para milisegundos

```
#include <sys/time.h>
#include <stdio.h>
#include <unistd.h>
int main() {
struct timeval start, end;
long mtime, seconds, useconds;

gettimeofday(&start, NULL);
usleep(2000);
gettimeofday(&end, NULL);

seconds = end.tv_sec - start.tv_sec;
useconds = end.tv_usec - start.tv_usec;

mtime = ((seconds) * 1000 + useconds/1000.0) + 0.5;
printf("Elapsed time: %ld milliseconds\n", mtime);
return 0;
}
```

Temporização na GPU
CUDA API

- CUDA oferece uma API para gravação de eventos que ocorrem na GPU.
  - A própria GPU grava o evento.
  - Evita interferência de eventos que estejam ocorrendo na CPU.
- Um **evento** em CUDA é essencialmente um *time stamp* na GPU que é gravado em um ponto do código determinado pelo usuário.

Temporização na GPU
CUDA API

- CUDA oferece uma API para gravação de eventos que ocorrem na GPU.
  - A própria GPU grava o evento.
  - Evita interferência de eventos que estejam ocorrendo na CPU.
- Um **evento** em CUDA é essencialmente um *time stamp* na GPU que é gravado em um ponto do código determinado pelo usuário.
- As funções da API para gerenciar eventos são:
  - **cudaEventCreate()**: criação do evento.
  - **cudaEventRecord()**: gravação do *time stamp* do evento.
  - **cudaEventSynchronize()**: espera que GPU termine a fila de trabalho.
  - **cudaEventElapsedTime()**: calcula tempo total em milisegundos entre dois eventos gravados.

Temporização na GPU

Exemplo: eventos em CUDA

```
...
float elapsedTime;
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

//realiza algum trabalho na GPU

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsedTime, &start, &stop);

printf("Tempo (ms) = \%f\n", elapsedTime);
...
```

Ocupação da GPU
Tópicos

- Definição e características
- Testes de desempenho
- *CUDA Occupancy Calculator*

Ocupação da GPU

- O escalonamento na GPU é baseado em unidades de processamento chamadas **warps**.
- Cada bloco é dividido em *warps* de 32 threads cada.
- Cada *warp* executa sequencialmente suas operações SIMD e é desescalonado quando algum operando estiver indisponível.
  - Neste caso, um novo *warp* do bloco será escalonado, o que mantém o SM ocupado.
- A **taxa de ocupação** de um kernel é o número de *warps* ativos sobre o máximo de *warps* suportados pelo dispositivo.
- A eficiência normalmente é garantida tendo-se um grande número de *warps* (e consequentemente de *threads*) em execução.

Ocupação da GPU

- Recursos compartilhados, como registradores e *shared memory* são os principais limitadores.
- Os parâmetros de lançamento do kernel, que indicam a quantidade de threads ativas, são determinantes.
- Porém, nem sempre uma melhor ocupação garante um melhor desempenho.
- Normalmente, existem limites mínimos para a taxa de ocupação.
- A partir de um certo ponto, o ganho em ocupação pode não ser tão importante para o tempo de execução.

Ocupação da GPU
Testes de desempenho

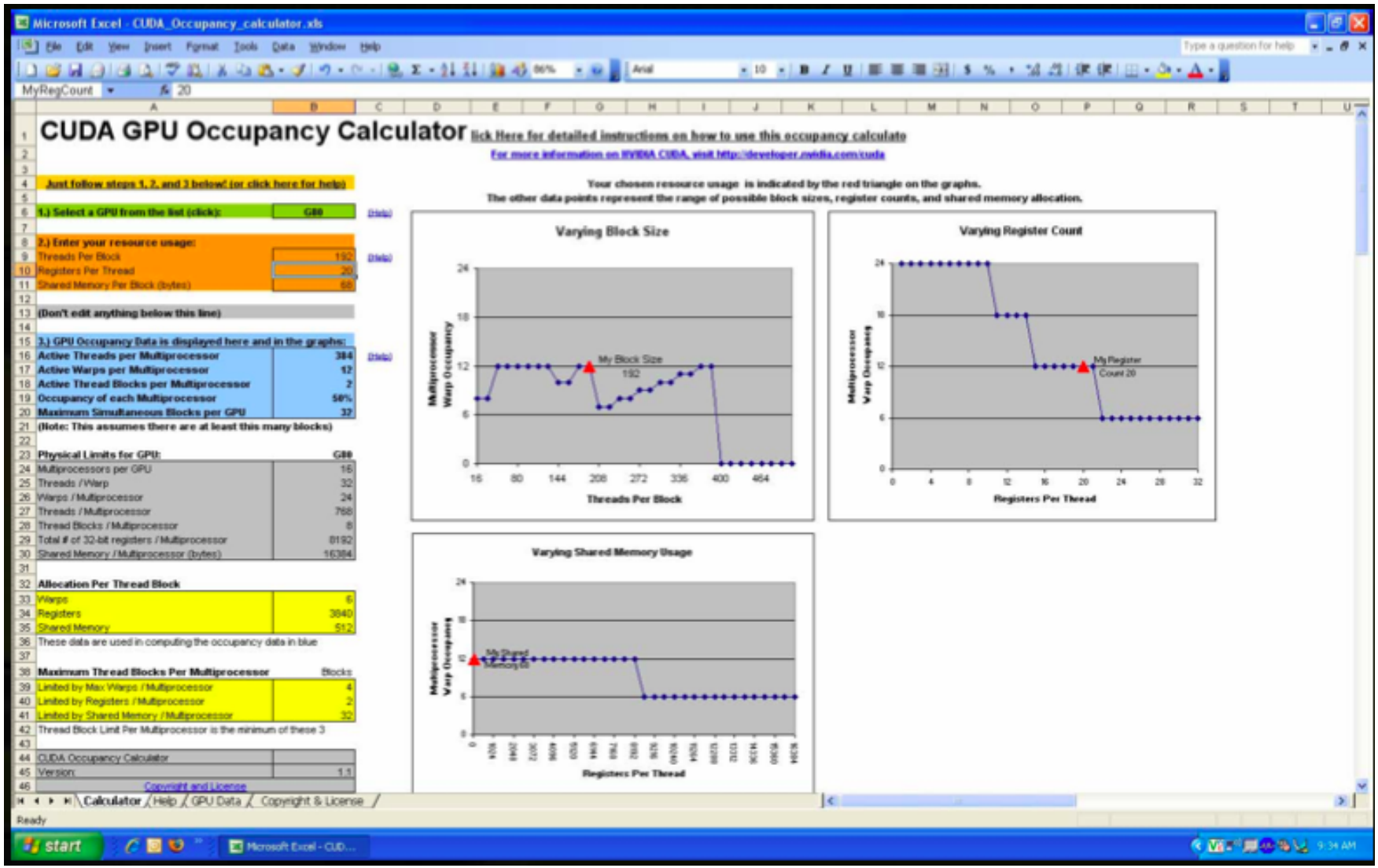
- A tabela a seguir mostra o comportamento da aplicação para cálculo do produto escalar presente no CUDA SDK.
- Foram usados diferentes dispositivos e tamanhos de blocos.
- As GPUs usadas (CUDA 2.3) foram:
  - GF 9800 GTX (capability 1.1 ou sm11)
  - GF GTX 295 (capability 1.3 ou sm13)

ScalarProd	**sm11**		**sm13**	
Block size	Occ(%)	T(ms)	Occ(%)	T(ms)
512	67	0.255	100	0.158
256	67	0.208	75	0.134
128	50	0.215	38	0.122
64	25	0.260	19	0.152
32	13	0.467	9	0.261

Ocupação da GPU
Testes de desempenho

- Mesmo com ocupacão igual ou até mesmo pior, blocos de 256 obtiveram melhor desempenho do que blocos de 512.
  - A existência de mais de um bloco evita paradas em caso de eventos de sincronização.
- Melhores escolhas para tamanho de bloco:
  - Blocos de 128 ou 256 threads (sempre múltiplos de 32)
  - Tamanho mínimo: 64 threads
  - Testar com a aplicação!
- Alguns limites mínimos sugeridos para a taxa de ocupação:
  - sm11: 25%
  - sm13: 18%

Ocupação da GPU
SDK Occupancy Calculator



Concorrência
Tópicos

- Recursos da arquitetura Fermi
- Execução assíncrona
- Memória não-paginada
- *Streams*
- *Driver API*

Execução concorrente e transferências de memória

Capability 2.x

- Execução concorrente de kernels:
  - Máximo 16 kernels concorrentes.
  - Kernels de um determinado contexto de CUDA não podem executar concorrentemente com kernels de um outro contexto.
  - O uso de muita memória pode ser um fator de limitação para a quantidade de kernels concorrentes.
- Transferências concorrentes:
  - *page-locked host* -> *device* pode ser feita ao mesmo tempo que *device* -> *page-locked host*.

Transferência entre *host* e *device*
Fermi e PCI Express

- Interface PCI Express V2 com 16 linhas.
- Permite que SMs acessem itens individuais na memória do *host*.
  - A Fermi pode endereçar diretamente e copiar em cache dados armazenados no *host*.
- Também permite a transferência assíncrona de grandes blocos de memória em alta velocidade através do mecanismo GigaThread SDT (*Streaming Data Transfer*).
- A Fermi pode processar uma transferência SDT para um *kernel* enquanto continua a executar outros *kernels*.
  - É possível explorar esta carcterística sobrepondo uma transferência com a execução de outros *kernels* que utilizem dados previamente carregados.

Execução concorrente assíncrona
Chamadas de funções assíncronas

- Facilitam a execução concorrente entre *host* e *device*.
- O controle retorna à thread principal (*host*) antes que o *device* tenha terminado o processamento.
- Os seguintes tipos de funções são assíncronas:
  - Lançamento de kernel.
  - Cópias de memória do tipo *device* -> *device*.
  - Cópias de memória do tipo *host* -> *device* (blocos de 64 KB ou menos).
  - Cópias de memória realizadas por funções com o sufixo *Async*.
  - Funções do tipo *memory set*.

Memória não-paginada (*page-locked/pinned*)

- Benefícios:
  - em alguns dispositivos os endereços podem ser mapeados diretamente no espaço de endereçamento do *device*, eliminando a necessidade de cópia explícita (*zero copy*).
  - as cópias entre *host* e *device* são automaticamente sobrepostas com a execução do kernel;
  - em sistemas com barramento do tipo *front side* a taxa de transferência é maior.

Memória não-paginada (*page-locked/pinned*)

- CUDA oferece as seguintes funções para alocação de memória não-paginada:
  - **cudaHostAlloc**() e **cudaFreeHost**(): alocam e liberam memória não-paginada no *host*.
  - **cudaHostRegister**(): utilizado para fazer o *lock* de um bloco de memória alocado com **malloc**().

Memória não-paginada (*page-locked/pinned*)

- CUDA oferece as seguintes funções para alocação de memória não-paginada:
  - **cudaHostAlloc**() e **cudaFreeHost**(): alocam e liberam memória não-paginada no *host*.
  - **cudaHostRegister**(): utilizado para fazer o *lock* de um bloco de memória alocado com **malloc**().
- Algumas restrições:
  - É um recurso limitado no *host*.
  - Pode diminuir o desempenho do sistema como um todo, visto que reduz o espaço de paginação.

Memória não-paginada (*page-locked/pinned*)
Flags

- Memória mapeada (*mapped*)
  - Permite que um bloco de memória alocada no *host* seja mapeada no espaço de endereçamento do *device*.
  - As transferências são realizadas implicitamente quando necessárias.
  - As transferências são automanticamente sobrepostas à execução do kernel
    - Não é necessário usar *streams* diferentes para isso, porém a sincronização correta é tarefa da aplicação.
  - Não pode ser usada juntamente com o espaço de endereçamento virtual unificado (UVA).

Memória não-paginada (*page-locked/pinned*)
Flags

- Memória mapeada (*mapped*)
  - Permite que um bloco de memória alocada no *host* seja mapeada no espaço de endereçamento do *device*.
  - As transferências são realizadas implicitamente quando necessárias.
  - As transferências são automanticamente sobrepostas à execução do kernel
    - Não é necessário usar *streams* diferentes para isso, porém a sincronização correta é tarefa da aplicação.
  - Não pode ser usada juntamente com o espaço de endereçamento virtual unificado (UVA).
- Memória *write-combining*
  - Libera as caches L1 e L2 do *host*.
  - O algoritmo de *snoop* não é aplicado, melhorando o desempenho da transferência em até 40% (segundo o *CUDA Programming Guide*).
  - Deve ser usada pelo *host* somente para escrita, pois a leitura é proibitivamente lenta.

Memória não-paginada (*page-locked/pinned*)

Exemplo: simpleZeroCopy (SDK) (1)

- O exemplo usa a função **cudaHostRegister()** que mapeia um bloco de memória alocado no host com **malloc()**.
- Para isso, a memória deve estar alinhada de acordo com o tamanho de uma página. O trecho abaixo mostra a macro utilizada no código do exemplo para realizar o alinhamento.

```
...
// Macro to aligned up to the memory size in question
#define MEMORY_ALIGNMENT  4096
#define ALIGN_UP(x,size)(((size_t)x+(size-1))&(~(size-1)))
...
```

Memória não-paginada (*page-locked/pinned*)
Exemplo: simpleZeroCopy (SDK) (2)

- O trecho abaixo aloca e mapeia os blocos de memória alocados no *host*.

```
...
/* Allow the retrieve of the device pointers*/
cudaSetDeviceFlags(cudaDeviceMapHost);

 /* Allocate mapped CPU memory. */
nelem = 1048576;
bytes = nelem*sizeof(float);
a_UA = (float *) malloc( bytes + MEMORY_ALIGNMENT );
b_UA = (float *) malloc( bytes + MEMORY_ALIGNMENT );
c_UA = (float *) malloc( bytes + MEMORY_ALIGNMENT );

/* Ensure memory is aligned to 4K (so we will need to padd memory accordingly)*/
a = (float *) ALIGN_UP( a_UA, MEMORY_ALIGNMENT );
b = (float *) ALIGN_UP( b_UA, MEMORY_ALIGNMENT );
c = (float *) ALIGN_UP( c_UA, MEMORY_ALIGNMENT );
cudaHostRegister(a, bytes, CU_MEMHOSTALLOC_DEVICEMAP);
cudaHostRegister(b, bytes, CU_MEMHOSTALLOC_DEVICEMAP);
cudaHostRegister(c, bytes, CU_MEMHOSTALLOC_DEVICEMAP);

/* Initialize the vectors. */
...
```

Memória não-paginada (*page-locked/pinned*)

Exemplo: simpleZeroCopy (SDK) (3)

- O trecho abaixo passa para o kernel os ponteiros mapeados na CPU.

```
...
 /* Get the device pointers for the pinned CPU memory mapped into t
     memory space. */
  cudaHostGetDevicePointer((void **)&d_a, (void *)a, 0);
  cudaHostGetDevicePointer((void **)&d_b, (void *)b, 0);
  cudaHostGetDevicePointer((void **)&d_c, (void *)c, 0);

 /* Call the GPU kernel using the CPU pointers residing in
    CPU mapped memory. */
  dim3 block(256);
  dim3 grid((unsigned int)ceil(nelem/(float)block.x));
  vectorAddGPU<<<grid, block>>>(d_a, d_b, d_c, nelem);
  cudaDeviceSynchronize();
/* Test and free the pointers */
...
```

Streams

- A concorrência em um dispositivo normalmente é obtida através do uso de *streams*.
- Um *stream* é uma sequência de comandos que executam em ordem e que podem partir de *threads* diferentes (do *host*).
- *Streams* diferentes são independentes e podem executar de forma concorrente entre si.
- Um objeto *stream* deve ser criado e utilizado como parâmetro para uma determinada sequência de lançamentos de kernel e cópias entre *host* e *device*.

Streams
Exemplo

- O trecho abaixo cria dois streams e aloca memória não-paginada.

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```

- O trecho abaixo executa operações sobrepostas nos dois streams.

```
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
    MyKernel<<<100, 512, 0, stream[i]>>>
          (outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
```

Streams
Sincronização

- Algumas funções podem ser usadas para sincronizar explicitamente as operações entre *streams*. Exemplos:
  - **cudaDeviceSynchronize**(): espera que todos os streams finalizem as operações correntes.
  - **cudaStreamSynchronize**(): espera o término das operações de um determinado *stream*.
- Sincronização implícita: algumas operações que apareçam intercaladas na *thread* do *host* não serão sobrepostas no *device*. Exemplos:
  - alocação de memória paginada no *host*;
  - alocação de memória no *device*;
  - cópia do tipo *device* -> *device*.

Driver API

- Faz parte da biblioteca dinâmica **nvcuda** e é instalada juntamente com os demais componentes do *driver* do dispositivo.
- Permite manipular objetos a partir de *handles* passados às funções.
- As funções da *driver API* possuem o prefixo **cu**.
- Permite a manipulação explícita de concorrência através do gerenciamento de múltiplos contextos.

Object	Handle	Description
Device	CUdevice	CUDA-enabled device
Context	CUcontext	Roughly equivalent to a CPU process
Module	CUmodule	Roughly equivalent to a dynamic library
Function	CUfunction	Kernel
Heap memory	CUdeviceptr	Pointer to device memory
CUDA array	CUarray	Opaque container for one-dimensional or two-dimensional data on the device, readable via texture or surface references
Texture reference	CUtexref	Object that describes how to interpret texture memory data
Surface reference	CUsurfref	Object that describes how to read or write CUDA arrays

Driver API
Contexto

- Um **contexto** em CUDA é análogo a um processo da CPU.
- Todas as ações da *driver* API estão encapsuladas num contexto.
- O sistema limpa automaticamente os recursos utilizados no momento em que o contexto é destruído.
- Cada contexto tem o seu próprio espaço de endereçamento.
- Uma thread no host pode ter apenas um contexto ativo de cada vez.

Driver API
Contexto

- Um **contexto** em CUDA é análogo a um processo da CPU.
- Todas as ações da *driver* API estão encapsuladas num contexto.
- O sistema limpa automaticamente os recursos utilizados no momento em que o contexto é destruído.
- Cada contexto tem o seu próprio espaço de endereçamento.
- Uma thread no host pode ter apenas um contexto ativo de cada vez.
- As principais funções para o gerenciamento de contexto são:
  - **cuCtxCreate**(): cria e empilha um novo contexto na pilha de contextos - o novo contexto passa a ser o atual.
  - **cuCtxPushCurrent() / cuCtxPopCurrent**(): empilha/desempilha um contexto existente.
  - **cuCtxAttach() / cuCtxDettach**(): altera o contexto atual incrementando/decrementando um contador de contextos.

Driver API
Exemplo: inicialização do contexto principal

- O trecho abaixo inicializa o contexto principal no *device 0*.

```
...
cuInit(0);
// Get handle for device 0
CUdevice cuDevice;
cuDeviceGet(&cuDevice, 0);
// Create context
CUcontext cuContext;
cuCtxCreate(&cuContext, 0, cuDevice);
...
```

Driver API
Exemplo: inicialização de um novo contexto

- O trecho abaixo inicializa e destroi um contexto dentro de uma *pthread*.
- Permite que exista uma *thread* por dispositivo instalado, cada uma com seu próprio contexto.

```
void* deviceDriver(DeviceParameters *deviceParams){

    /* Declare the handle for the new context.
        Some driver API functions will require this handle */
    CUcontext   hcuContext;
    /* Declare other variables...*/

    /* cuCtxCreate: Function works on floating contexts
       and current context */
    cuCtxCreate(&hcuContext, 0, deviceParams->hcuDevice);
    //obs: could be any of available devices

    ...

    /* at the end of the thread, destroy the context for this device
        and return to the main context */
     cuCtxDestroy (hcuContext);
}
```

Multi-GPU
Tópicos

- *Unified Virtual Address Space* (UVA)
- Seleção de dispositivo
- Acesso à memória *peer-to-peer*
- Cópia de memória *peer-to-peer*

Sistemas *multi-device* (ou multi-GPU)

- É muito comum encontrar sistemas com mais de uma GPU.
- CUDA 4.0 incluiu diversas funcionalidades para facilitar o desenvolvimento em tais sistemas.
- Entre os benefícios incluídos encontram-se:
  - Escolha do dispositivo independente de contexto.
  - Cópia de memória entre dispositivos, sem a necessidade de usar a memória do *host* como intermediária.
  - Acesso direto à memória mapeada ou ao *Unified Virtual Addres Space* (UVA) entre dispositivos (*capability* 2.0 ou maior).

Unified Virtual Addres Space (UVA)

- É um espaço de endereçamento virtual único para *host* e demais dispositivos presentes no sistema.
- Existe em sistemas de 64 bits e *capability* 2.0 ou maior.
- É criado no *host* com a função **cudaHostAlloc()** (ou **cudaMallocHost()**) e no(s) *device(s)* com **cudaAlloc*()**.
- A função **cudaPointerGetAttributes()** indica o tipo de memória de um determinado ponteiro.
- Acessos do tipo *peer-to-peer* entre dois dispositivos que usem UVA não precisam ser mapeados entre eles.

Seleção de dispositivo

- O dispositivo a ser utilizado pode ser selecionado a qualquer momento através da função **cudaSetDevice()**.
- Exemplo:

```
size_t size = 1024 * sizeof(float);
cudaSetDevice(0);              // Set device 0 as current
float* p0;
cudaMalloc(&p0, size);         // Allocate memory on device 0
MyKernel<<<1000, 128>>>(p0); // Launch kernel on device 0
cudaSetDevice(1);              // Set device 1 as current
float* p1;
cudaMalloc(&p1, size);         // Allocate memory on device 1
MyKernel<<<1000, 128>>>(p1); // Launch kernel on device 1
```

Acesso à memória (*peer-to-peer*)

- Dispositivos com *capability* 2.0 ou maior podem endereçar a memória de outros dispositivos similares.
- A função **cudaDeviceEnablePeerAccess()** deve ser usada para habilitar esta funcionalidade.
- Exemplo (utilizando UVA):

```
cudaSetDevice(0);                       // Set device 0 as current
float* p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size);                  // Allocate memory on device 0
MyKernel<<<1000, 128>>>(p0);            // Launch kernel on device 0
cudaSetDevice(1);                       // Set device 1 as current
cudaDeviceEnablePeerAccess(0, 0);       // Enable peer-to-peer access
                                        // with device 0

// Launch kernel on device 1
// This kernel launch can access memory on device 0 at address p0
MyKernel<<<1000, 128>>>(p0);
```

Acesso à memória (*peer-to-peer*)

- A memória mapeada pode ser utilizada através das funções **cudaPeerRegister()** e **cudaPeerGetDevicePointer()**.
- Exemplo (utilizando memória mapeada):

```
cudaSetDevice(0);                       // Set device 0 as current
float* p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size);                  // Allocate memory on device 0
MyKernel<<<1000, 128>>>(p0);            // Launch kernel on device 0
cudaSetDevice(1);                       // Set device 1 as current
```

Acesso à memória (*peer-to-peer*)

- Exemplo (utilizando memória mapeada - continuação):

```
cudaDeviceEnablePeerAccess(0, 0); // Enable peer-to-peer access
                                  // with device 0
cudaPeerRegister(
   p0,                            // Register pointer p0
   0,                             // pointing to memory of device 0
                                  // for access by the current device
                                  // (i.e. device 1 in this case)
   cudaPeerRegisterMapped);       // directly via a device pointer
float* p0_for_1;
cudaPeerGetDevicePointer(
   &p0_for_1,      // Get the device pointer for the current device
                   // (i.e. device 1 in this case)
   p0,             // for p0
   0);             // pointing to memory of device 0

// Launch kernel on device 1
// This kernel launch can access memory on device 0 at address p0
// via the pointer p0_for_1
MyKernel<<<1000, 128>>>(p0_for_1);
```

Cópia de memória (*peer-to-peer*)

- As funções **cudaMemcpyPeer(), cudaMemcpyPeerAsync(), cudaMemcpy3DPeer()** ou **cudaMemcpy3DPeerAsync()** podem ser usadas para copiar dados entre dispositivos.
  - O acesso direto *peer-to-peer* torna a cópia desnecessária na maioria dos casos e é mais rápido.
- Exemplo:

```
cudaSetDevice(0);                     // Set device 0 as current
float* p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size);                // Allocate memory on device 0
cudaSetDevice(1);                     // Set device 1 as current
float* p1;
cudaMalloc(&p1, size);                // Allocate memory on device 1
cudaSetDevice(0);                     // Set device 0 as current
MyKernel<<<1000, 128>>>(p0);          // Launch kernel on device 0
cudaSetDevice(1);                     // Set device 1 as current
cudaMemcpyPeer(p1, 1, p0, 0, size); // Copy p0 to p1
MyKernel<<<1000, 128>>>(p1);          // Launch kernel on device 1
```

Compute Visual Profiler
Tópicos

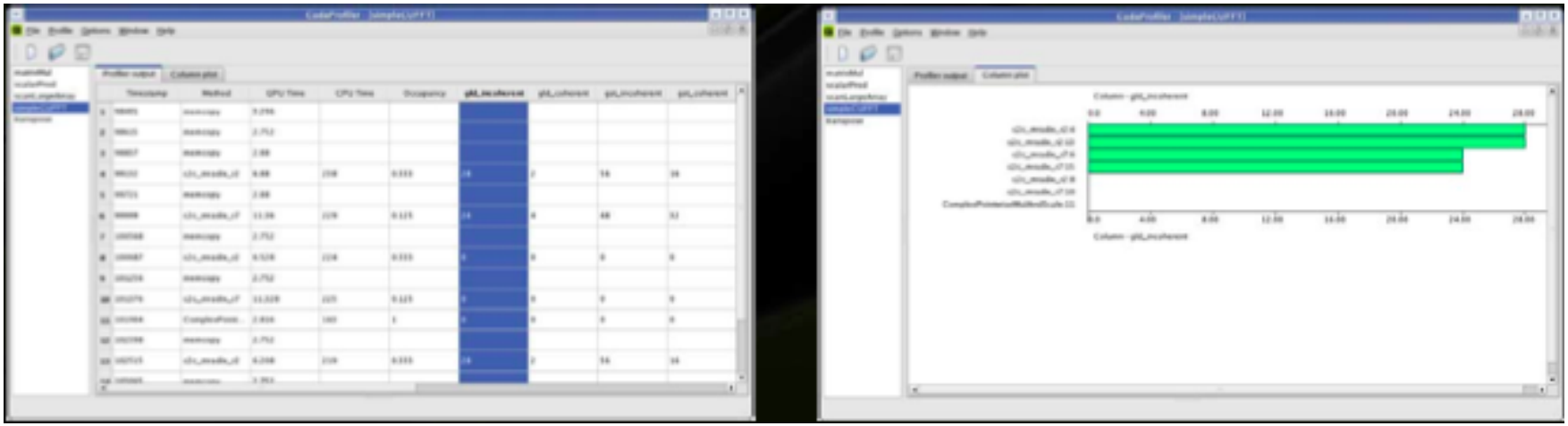
- Ferramentas de auxílio à programação
- Principais características do *Compute Visual Profiler*
- Tipos de sinais capturados
- Interpretação dos contadores

Ferramentas de auxílio à programação

- As ferramentas de auxílio à programação CUDA podem ser bastante úteis, principalmente com relação à análise de desempenho.
- Entre as ferramentas disponíveis destacam-se:
  - NVIDIA Compute Visual Profiler
  - Microsoft Parallel Nsight
- Além disso, existe também uma planilha disponibilizada no SDK da NVIDIA para cálculo da taxa de ocupação:
  - NVIDIA Occupancy Calculator

Compute Visual Profiler

- É uma ferramenta com interface gráfica que apresenta dados obtidos durante a execução de uma aplicação em CUDA.
- É instalada juntamente com o *toolkit* de CUDA 4.0.
- Apresenta dados de *profiling* na forma de tabelas e alguns gráficos baseados principalmente em contadores implementados em hardware.



Tipos de sinais capturados e contadores

- Temporização
  - **timestamp**
- Acessos à memória local (*loads/stores*):
  - **local_load**, **local_store**
- Total de desvios e desvios divergentes:
  - **branch**, **divergent_branch**
- Contador de instruções:
  - **instructions**
- Serialização de *warps* por conflitos de acesso à memória (compartilhada ou constante):
  - **warp_serialize**
- Contador de blocos executados
  - **cta_launched**

Interpretação dos contadores

- Os valores representam eventos dentro de um *warp*.
- Avaliam apenas um único SM.
  - Valores não correspondem ao número total de *warps* lançados para um determinado kernel.
  - Deve-se lançar uma quantidade suficiente de blocos para assegurar que o SM alvo possua uma quantidade significativa do trabalho total.
- Os valores serão melhor aproveitados se utilizados para identificar um desempenho relativo entre uma versão não-otimizada e uma versão otimizada de código.

Considerações finais
Recursos a serem utilizados

- Memória:
  - Uso de memória não-paginada mapeada.
  - UVA (*Unified Virtual Address Space*)
- Sobreposição de operações:
  - Funções assíncronas (transferências de memória e execução de kernel).
  - Uso de *streams*.
- Multi-GPU:
  - Seleção de dispositivo independente de contexto.
  - Uso de contextos para gerenciar dispositivos a partir de diferentes threads do host (*Driver API*).
  - Acesso à memória *peer-to-peer*
- Uso de ferramentas de análise:
  - *CUDA Occupancy Calculator*
  - *Compute Visual Profiler*

Bibliografia

- Kirk, D. B., Hwu, W.W., Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufman, 2010.
- Sanders, J.; Kandrot, E. CUDA by Example - An Introduction to General-Purpose GPU Programming. Adison-Wesley, 2011
- NVIDIA Corporation, NVIDIA CUDA API Reference Manual 4.0, 2011.
- NVIDIA Corporation, NVIDIA CUDA C Programming Guide - 4.0, 2011.
- NVIDIA Corporation, NVIDIA CUDA C Best Practices Guide - 4.0, 2011

Bibliografia

- Sutter, H. The Free Lunch Is Over - A Fundamental Turn Toward Concurrency in Software Dr. Dobb's Journal, 30(3), Mar 2005
- Borkar, S.; Chien, A. A. 2011. The future of microprocessors. Commun. ACM 54, 5 (May 2011), 67-77. DOI=10.1145/1941487.1941507 http://doi.acm.org/10.1145/1941487.1941507
- Volkov, V. Better Performance at Lower Occupancy. GTC 2010 (presentation available in http://www.nvidia.com/object/gtc2010-presentation-archive.html).