Tópicos Especiais Em Redes E Sistemas Distribuídos: Programação Paralela

- NVIDIA - GPU Teaching Kit, accelerated computing
- Sylvain Collange, GPU Programming
  http://homepages.dcc.ufmg.br/~sylvain.collange/gpucourse
- Manuel Ujaldón, CUDA course

Taxonomia de Flynn

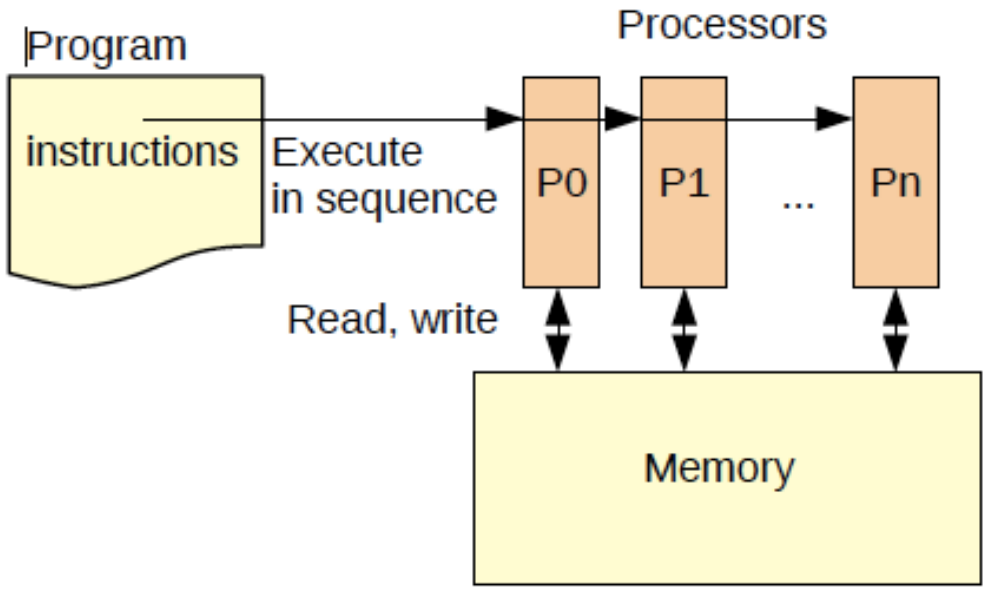
- Eixos: instruções por ciclo de relógio x fluxo de dados usados como entrada durante um ciclo de relógio

	Single Instruction	Multiple Instruction
Single Data	**SISD (Von Newmann)** Computador Serial (não-paralelo). Exec. deterministica	**MISD** Um único fluxo de dados é processado por múltiplas unidades em cada ciclo (cripto cracking, tolerância a falhas)
Multiple Data	**SIMD (Cray-1, Processadores vetoriais, GPU?)** Todas as unidades de processamento executam a mesma instrução em cada ciclo de relógio x Cada unidade de processamento pode operar sobre diferentes elementos de dado	**MIMD** A maioria dos supercomputadores modernos, clusters, computadores multi-processadores SMP, PCs multi-core

Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966

Modelo Paralelo : PRAM

- 1 memória, tempo de acesso constante
- N processadores (para problema de tamanho N)
- Passos síncronos
  - Todos os processadores executam uma instrução simultaneamente
  - Usualmente a mesma instrução
- Cada processador sabe seu próprio número



S. Fortune and J. Wyllie. Parallelism in random access machines. Proc. ACM symposium on Theory of computing. 1978

PRAM example: SAXPY

Given vectors X and Y of size n, compute vector R←a*X+Y with n processors

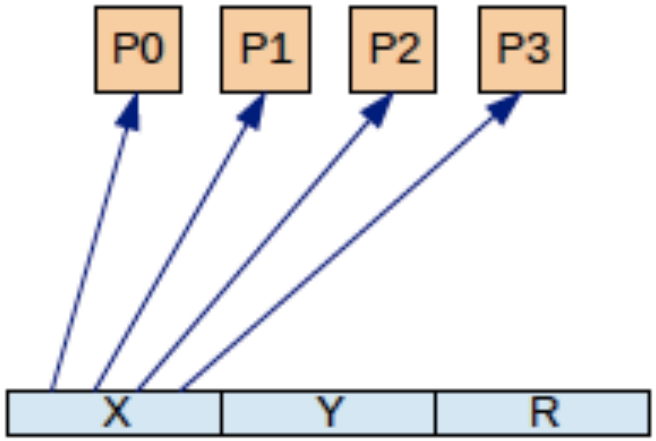
Processor Pi:

xi ← X[i]

yi ← Y[i]

ri ← a * xi + yi

R[i] ← ri



PRAM example: SAXPY

Given vectors X and Y of size n, compute vector R←a*X+Y with n processors

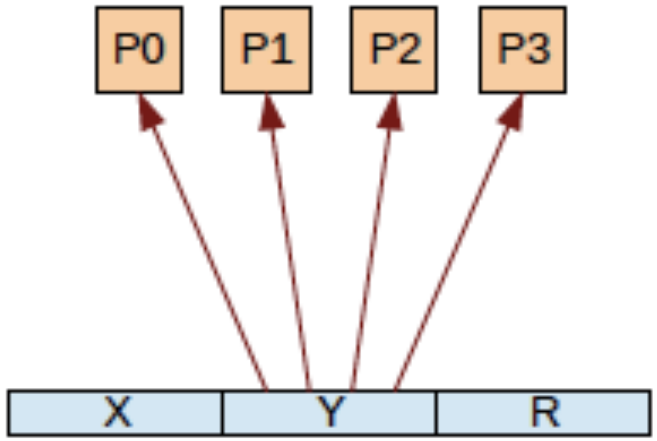
Processor Pi:

xi ← X[i]

yi ← Y[i]

ri ← a * xi + yi

R[i] ← ri



PRAM example: SAXPY

Given vectors X and Y of size n, compute vector R←a*X+Y with n processors

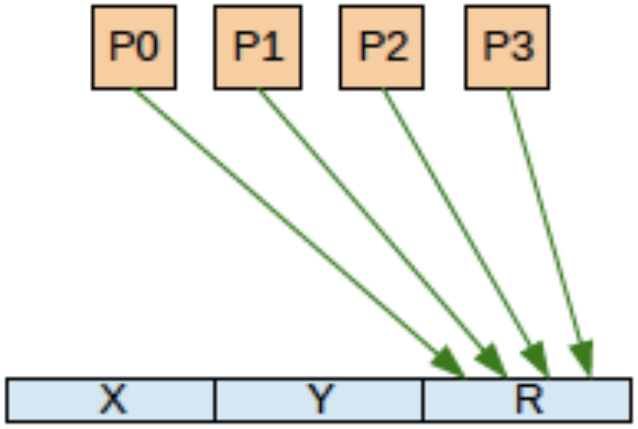
Processor Pi:

xi ← X[i]

yi ← Y[i]

ri ← **a** * **xi** + **yi**

R[i] ← ri



PRAM example: SAXPY

Given vectors X and Y of size n, compute vector R←a*X+Y with n processors

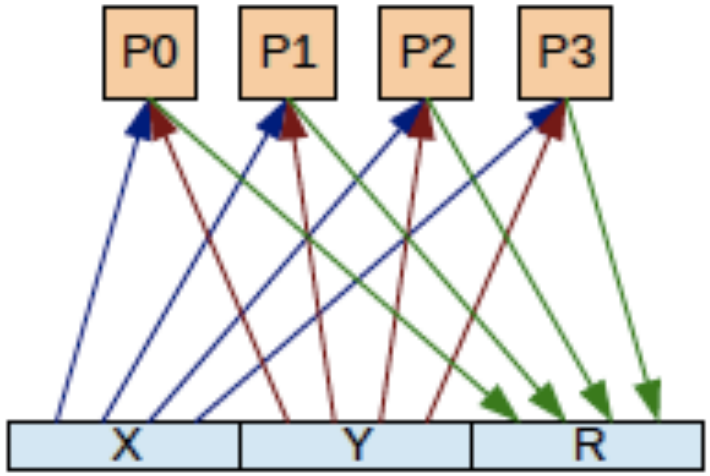
Processor Pi:

xi ← X[i]

yi ← Y[i]

ri ← **a** * **xi** + **yi**

R[i] ← ri



Complexity: O(1) with n processors

O(n) with 1 processor

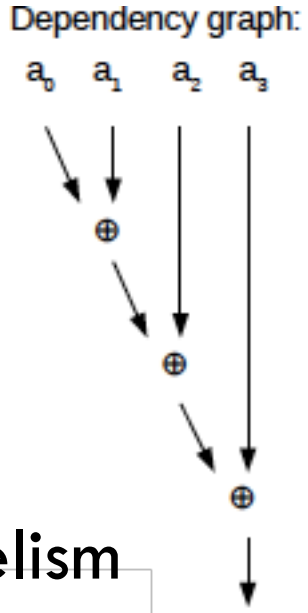
Redução: exemplo

- Given a vector a of size n, compute the sum of elements with n/2 processors
- Sequential algorithm

  for i = 0 to n-1

  sum ← sum + a[i]

  - Complexity: O(n) with 1 processor
- All operations are dependent: no parallelism
  - Need to use associativity to reorder the sum

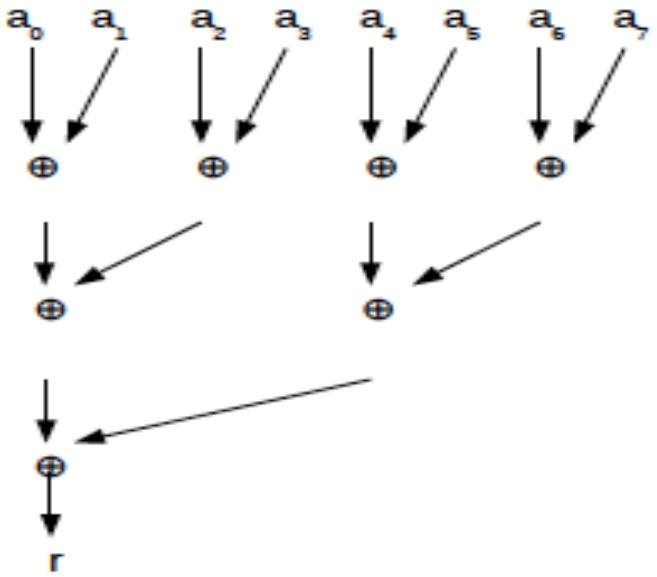


Redução Paralela: grafo de dependencias

- Calculamos r=(...((a0⊕a1)⊕(a2 ⊕a3))⊕...⊕an-1)...)

Pares

Grupos de 4

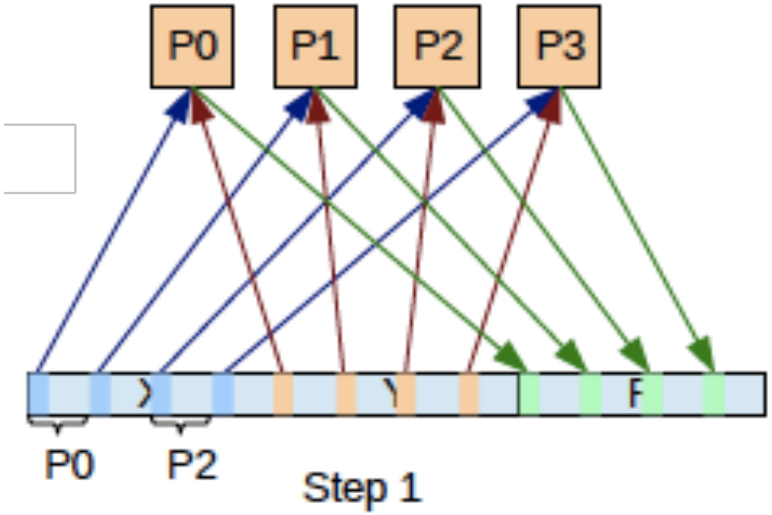


Menos do que n processadores

- A computer that grows with problem size is not very realistic
- Assume p processors for n elements, p < n
- Back to R←a^xX+Y example
- Solution 1: group by blocks

assign ranges to each processor

Processor Pi:
for j = i* ⌈n/p⌉ to (i+1)*⌈n/p⌉-1
xj ← X[j]
yj ← Y[j]
rj ← a x xj + yj
R[j] ← rj

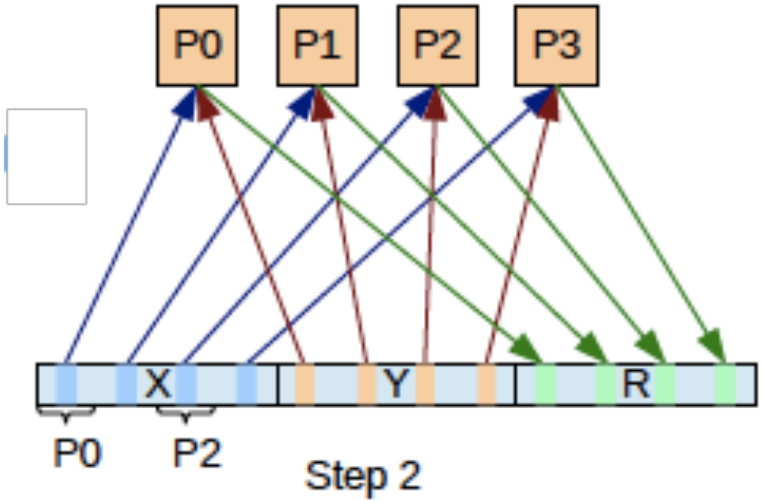


Menos do que n processadores

- A computer that grows with problem size is not very realistic
- Assume p processors for n elements, p < n
- Back to R←a^xX+Y example
- Solution 1: group by blocks

assign ranges to each processor

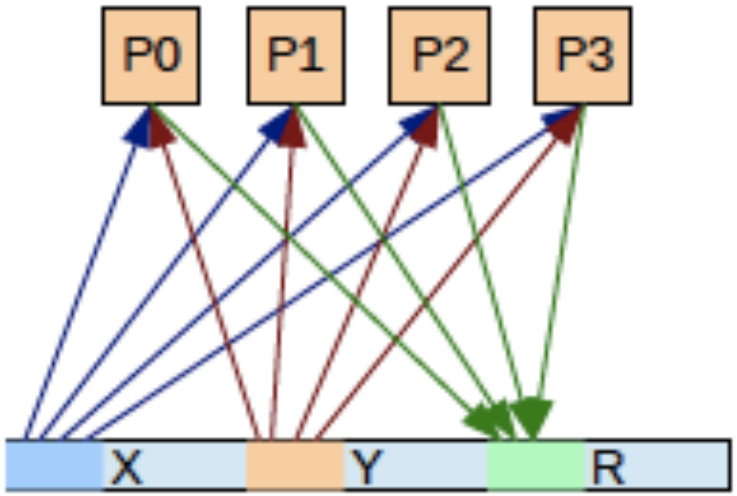
Processor Pi:
for j = i* ⌈n/p⌉ to (i+1)*⌈n/p⌉-1
xj ← X[j]
yj ← Y[j]
rj ← a x xj + yj
R[j] ← rj



Solution 2: slice and interleave

Processor Pi:

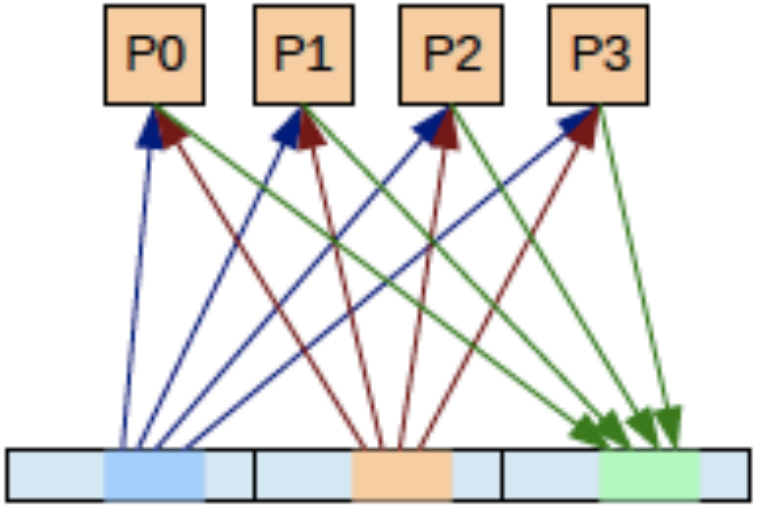
```
for j = i to n step p
    xj ← X[j]
    yj ← Y[j]
    rj ← a x xj + yj
    R[j] ← rj
```



Step 1

Solution 2: slice and interleave

Processor Pi:
for j = i to n step p
xj ← X[j]
yj ← Y[j]
rj ← a x xj + yj
R[j] ← rj



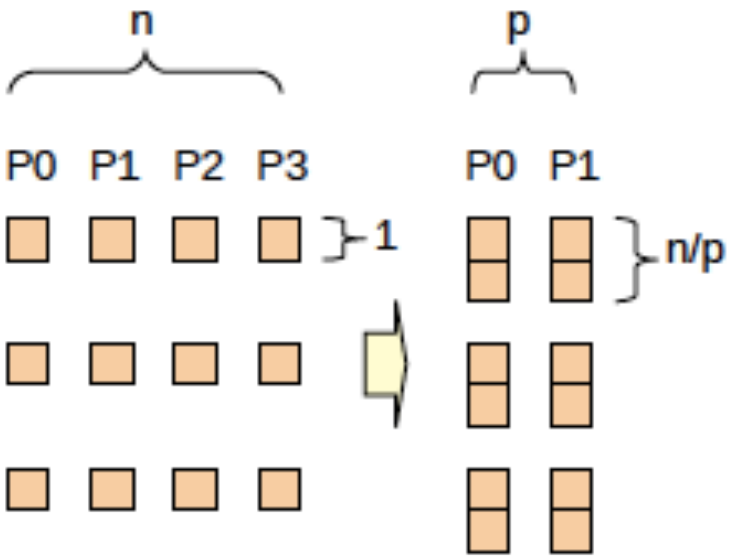
Step 2

Equivalent in the PRAM model

But not in actual programming!

Brent's theorem

- Generalization for any PRAM algorithm of complexity C(n) with n processors
- Derive an algorithm for n elements, with p processors only
- Idea: each processor emulates n/p virtual processors
  - Each emulated step takes n/p actual steps
- Complexity: O(n/p * C(n))



GPU Teaching Kit

Accelerated Computing

Lecture 2.2 - Introduction to CUDA C

Memory Allocation and Data Movement API Functions

NVCC Compiler

- NVIDIA provides a CUDA-C compiler
  - nvcc
- NVCC compiles device code then forwards code on to the host compiler (e.g. g++)
- Can be used to compile & link host only applications

Trabalho de laboratório

1. Login na 200.131.37.129 com seu usuário e senha.
2. O trabalho de hoje é uma introdução à programação em CUDA.
3. Compile e execute os exemplos a seguir.

Example 1: Hello World

```
int main() {
   printf("Hello World!\n");
   return 0;
}
```

Instructions:

1. Build and run the hello world code
2. Modify Makefile to use nvcc instead of g++
3. Rebuild and run

CUDA Example 1: Hello World

```
__global__ void mykernel(void) {
}

int main(void) {
   mykernel<<<1,1>>>();
   printf("Hello World!\n");
   return 0;
}
```

Instructions:

1. Add kernel and kernel launch to main.cu
2. Try to build

More on CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` defines a kernel function
  - Each "__" consists of two underscore characters
  - A kernel function must return `void`
- `__device__` and `__host__` can be used together
- `__host__` is optional if used alone

CUDA Example 1: Build Considerations

- Build failed
  - Nvcc only parses .cu files for CUDA
- Fixes:
  - Rename main.cc to main.cu

  OR
  - nvcc –x cu
    - Treat all input files as .cu files

Instructions:
1. Rename main.cc to main.cu
2. Rebuild and Run

Hello World! with Device Code

```
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

Output:

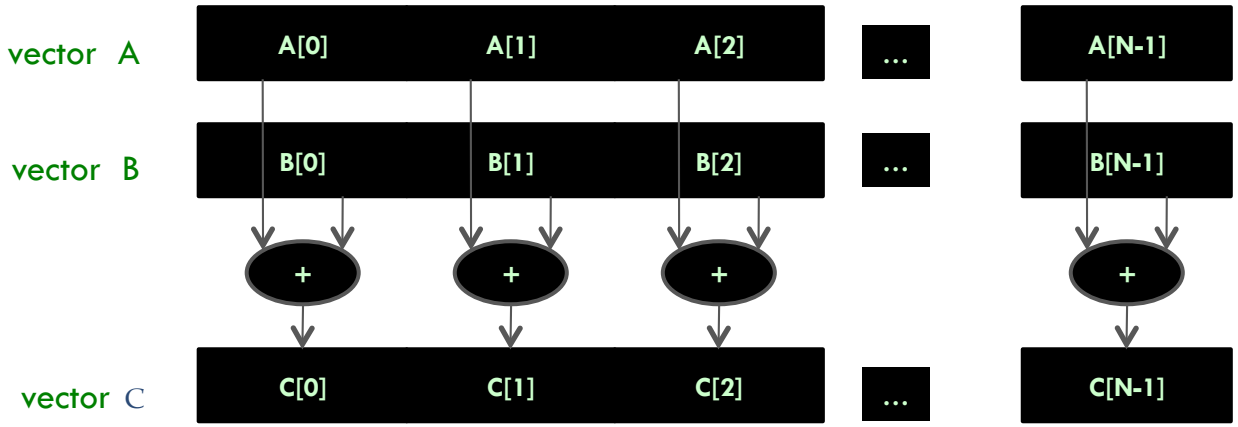
```
$ nvcc main.cu
$ ./a.out
Hello World!
```

- `mykernel` (does nothing, somewhat anticlimactic!)

Objective

- To learn the basic API functions in CUDA host code
  - Device Memory Allocation
  - Host-Device Data Transfer

Data Parallelism - Vector Addition Example



Vector Addition – Traditional C Code

```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i<n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
     // I/O to read h_A and h_B, N elements
     …
    vecAdd(h_A, h_B, h_C, N);
}
```

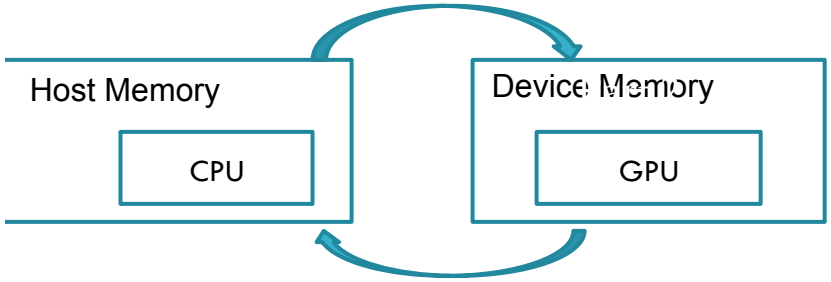
Testar código na GPU

```
// Device code:
//Compute vector sum C = A + B
__global__ void vecAdd(float *h_A,
float *h_B, float *h_C)
{
    int id = threadIdx.x;
    //Coloque seu código aqui.
}
```

```
// Host code:
int main() {
   int size = n * sizeof(float); float *d_A, *d_B, *d_C;
   //Inicialize os vetores A, B e C!!

   cudaMalloc((void **) &d_A, size);
   cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_C, size);

    vecAdd<<<1,32>>>(d_A, d_B, d_C);

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    //imprima o vetor C ou compare-o com a saida da CPU
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

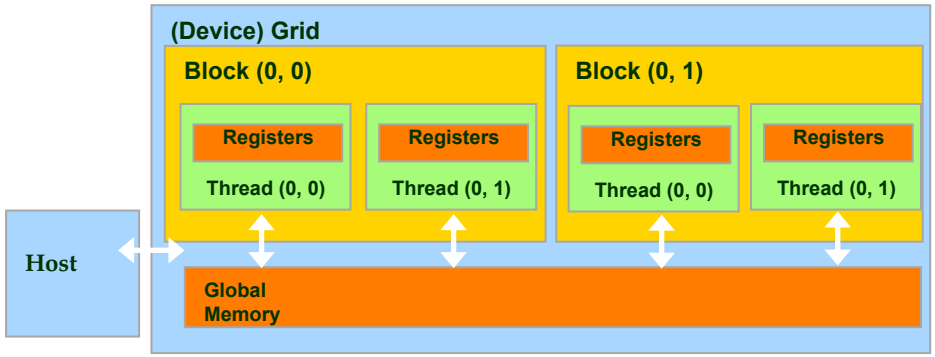
Heterogeneous Computing vecAdd CUDA Host Code

Part 1




```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n* sizeof(float);
    float *d_A, *d_B, *d_C;
    // Part 1
    // Allocate device memory for A, B, and C
    // copy A and B to device memory

    // Part 2
    // Kernel launch code – the device performs the actual vector addition

    // Part 3
    // copy C from the device memory

}
```

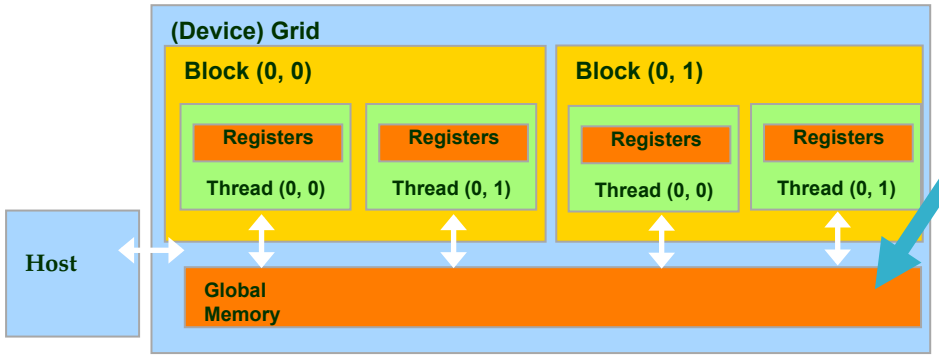
Partial Overview of CUDA Memories



- Device code can:
  - R/W per-thread registers
  - R/W all-shared global memory
- Host code can
  - Transfer data to/from per grid global memory

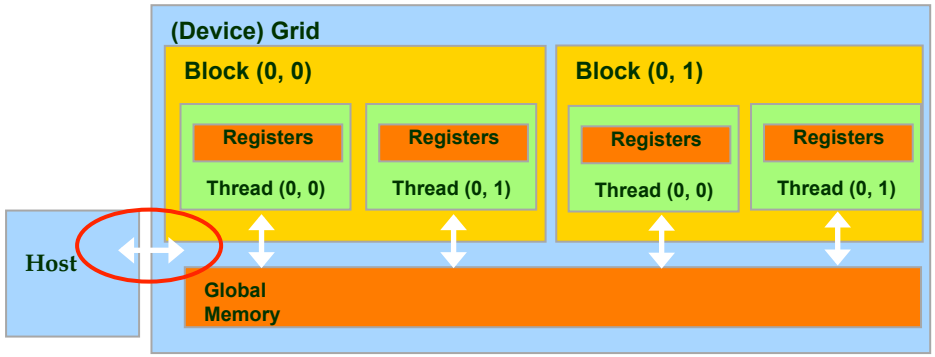
We will cover more memory types and more sophisticated memory models later.

CUDA Device Memory Management API functions



- cudaMalloc()
  - Allocates an object in the device global memory
  - Two parameters
    - **Address of a pointe**r to the allocated object
    - **Size of** allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
  - One parameter
    - **Pointer** to freed object

Host-Device Data Transfer API functions



- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer
  - Transfer to device is asynchronous

Vector Addition Host Code

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
     cudaMalloc((void **) &d_B, size);
     cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
     cudaMalloc((void **) &d_C, size);

     // Kernel invocation code
     cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
     cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

In Practice, Check for API Errors in Host Code

```
cudaError_t  err = cudaMalloc((void **) &d_A, size);

if (err != cudaSuccess)  {
   printf("%s in %s at line %d\n",   cudaGetErrorString(err), __FILE__,
   __LINE__);
   exit(EXIT_FAILURE);
}
```

Código com verificação de erros e tempo de execução

```
#ifndef checkCudaErrors
static void HandleError( cudaError_t err, const char *file, int line ) {
    if (err != cudaSuccess) {
        printf( "%s in %s at line %d\n", cudaGetErrorString( err ),
                file, line );
        exit( EXIT_FAILURE );
    }
}
#define checkCudaErrors( err ) (HandleError( err, __FILE__, __LINE__ ))
#endif
```

Exemplo:

```
checkCudaErrors(cudaMalloc((void**)&d_A, bytes_N));
```

Trabalho: conheça sua GPU

- deviceQuery

The output of cudaGetDeviceProperties

- This is exactly the output you get from the "DeviceQuery" code in the CUDA SDK.

```
There are 4 devices supporting CUDA

Device 0: "GeForce GTX 480"
  CUDA Driver Version:                           4.0
  CUDA Runtime Version:                          4.0
  CUDA Capability Major revision number:         2
  CUDA Capability Minor revision number:         0
  Total amount of global memory:                 1609760768 bytes
  Number of multiprocessors:                     15
  Number of cores:                               480
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Clock rate:                                    1.40 GHz
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     No
  Integrated:                                    No
  Support host page-locked memory mapping:       Yes
  Compute mode:                                  Default (multiple host threads can use this device simultaneously)
  Concurrent kernel execution:                   Yes
  Device has ECC support enabled:                No
```

Tempo de execução

```
cudaEvent_t     start, stop;
  HANDLE_ERROR( cudaEventCreate( &start ) );
  HANDLE_ERROR( cudaEventCreate( &stop ) );
  HANDLE_ERROR( cudaEventRecord( start, 0 ) );
kernel<<<grids,threads>>>(args);//alocação, chamada ao kernel, copia de dados
  HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
  HANDLE_ERROR( cudaEventSynchronize( stop ) );
  float   elapsedTime;
  HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                      start, stop ) );
  printf( "Time to generate:  %3.1f ms\n", elapsedTime );

  HANDLE_ERROR( cudaEventDestroy( start ) );
  HANDLE_ERROR( cudaEventDestroy( stop ) );
```

GPU Teaching Kit

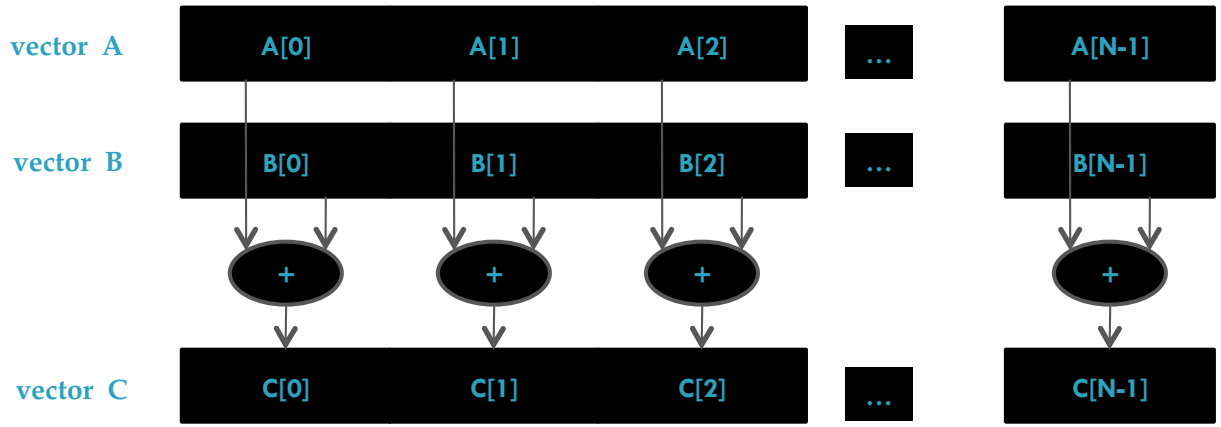
Lecture 2.3 – Introduction to CUDA C

Threads and Kernel Functions

Objective

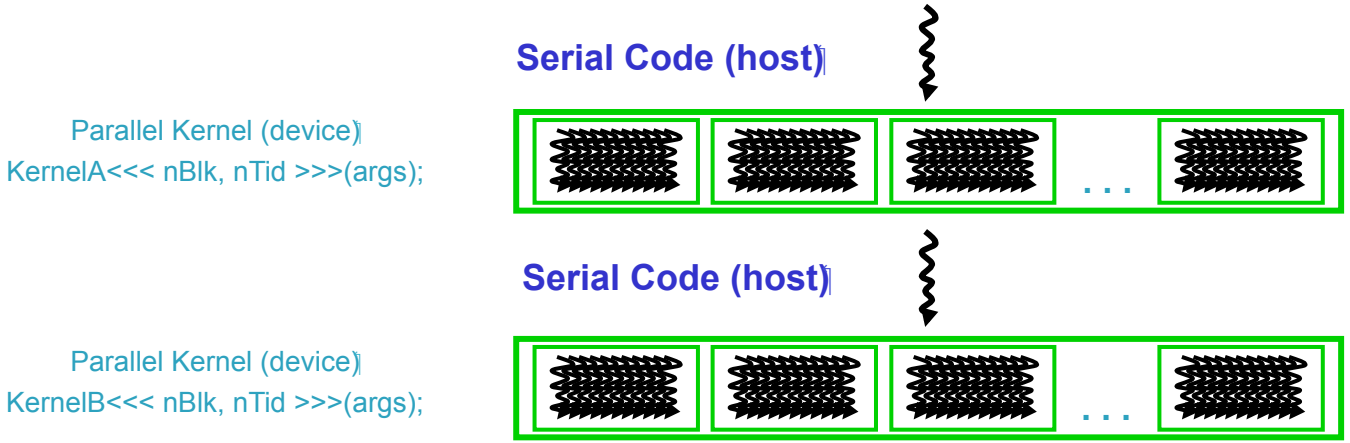
- To learn about CUDA threads, the main mechanism for exploiting of data parallelism
  - Hierarchical thread organization
  - Launching parallel execution
  - Thread index to data index mapping

Data Parallelism - Vector Addition Example

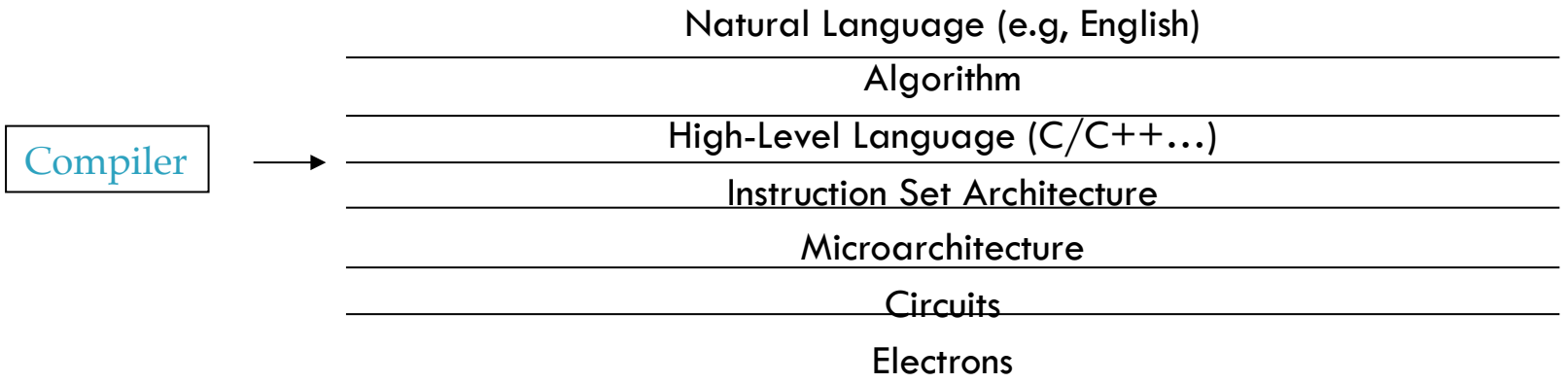


CUDA Execution Model

- Heterogeneous host (CPU) + device (GPU) application C program
  - Serial parts in **host** C code
  - Parallel parts in **device** SPMD kernel code



From Natural Language to Electrons

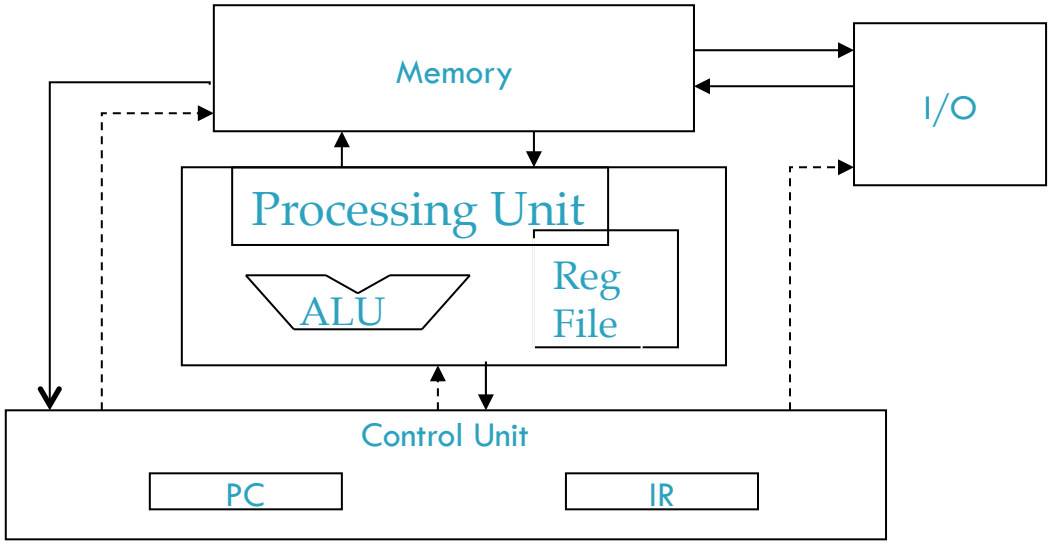


A program at the ISA level

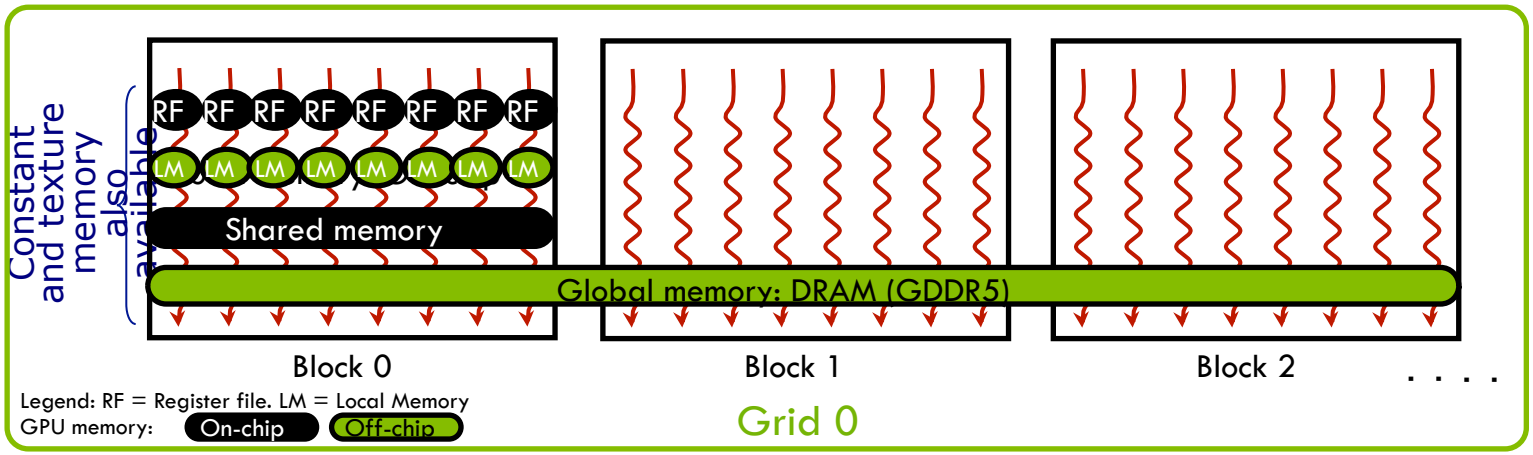
- A program is a set of instructions stored in memory that can be read, interpreted, and executed by the hardware.
  - Both CPUs and GPUs are designed based on (different) instruction sets
- Program instructions operate on data stored in memory and/or registers.

A Thread as a Von-Neumann Processor

A thread is a “virtualized” or “abstracted”
Von-Neumann Processor



GPU memory: Scope and location

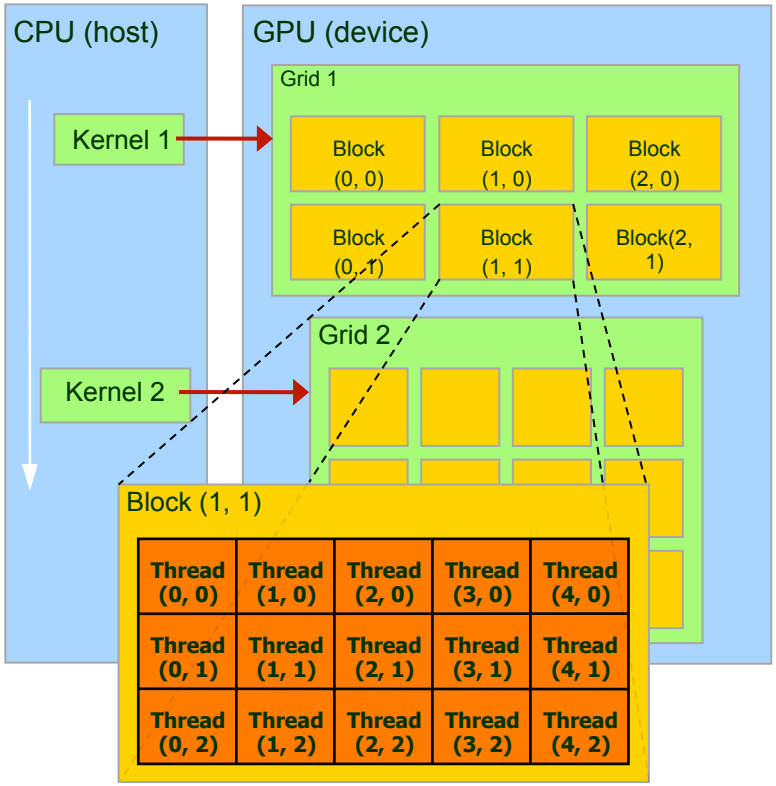


Blocks to share the same multiprocessor if memory constraints are fulfilled

- Threads within a block can use the shared memory to perform tasks in a more cooperative and faster manner.
- Global memory is the only visible to threads, blocks and kernels.

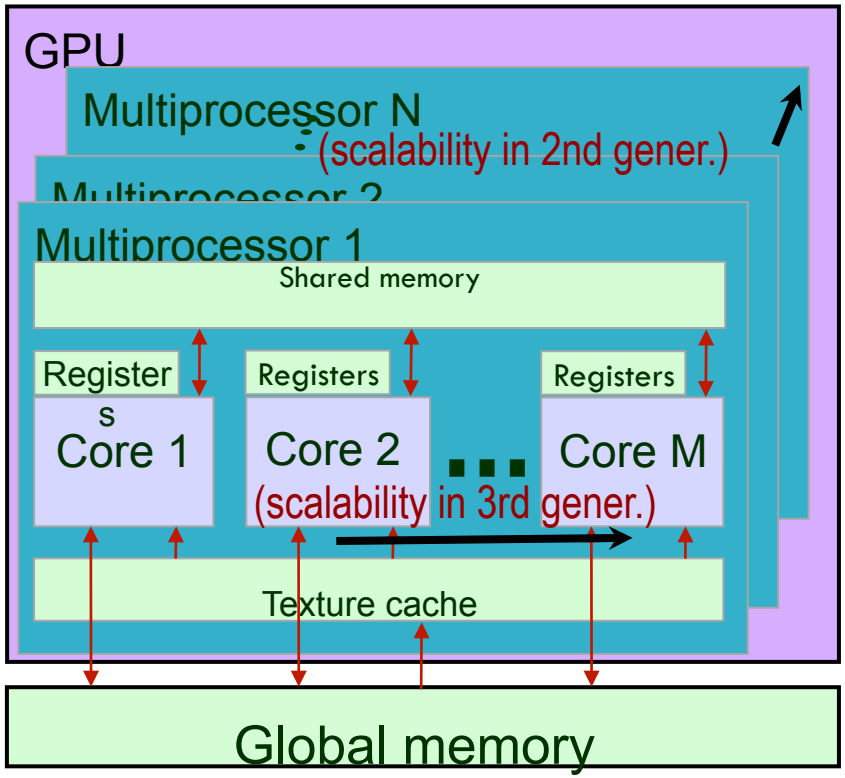
Partitioning data and computations

- A block is a batch of threads which can cooperate by:
- Sharing data via shared memory.
- Synchronizing their execution for hazard-free memory accesses.
- A kernel is executed as a 1D or 2D grid of 1D, 2D or 3D of thread blocks.
- Multidimensional IDs are very convenient when addressing multidimensional arrays, for each thread has to bound its area/volume of local computation.



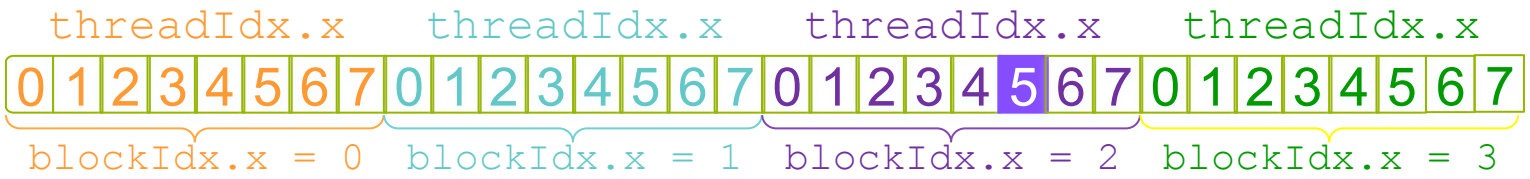
Running in parallel (regardless of hardware generation)

- `vecAdd <<< 1, 1 >>>`
  ( ) Executes 1 block composed of 1 thread - no parallelism.
- `vecAdd <<< B, 1 >>>`
  ( ) Executes B blocks composed on 1 thread. Inter-multiprocessor parallelism.
- `vecAdd <<< B, M >>>`
  ( ) Executes B blocks composed of M threads each. Inter- and intra-multiprocessor parallelism.



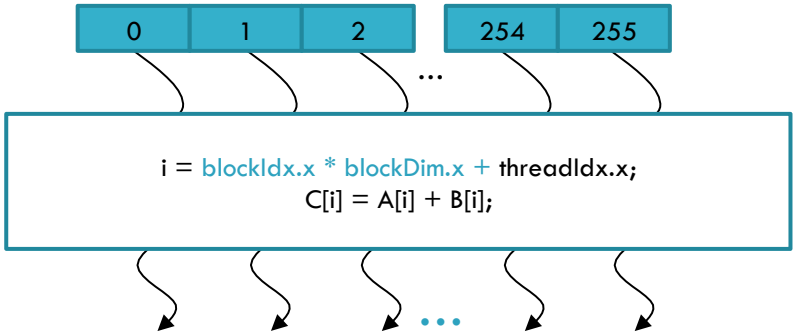
Indexing arrays with blocks and threads

Qual seria a configuração nesse caso?

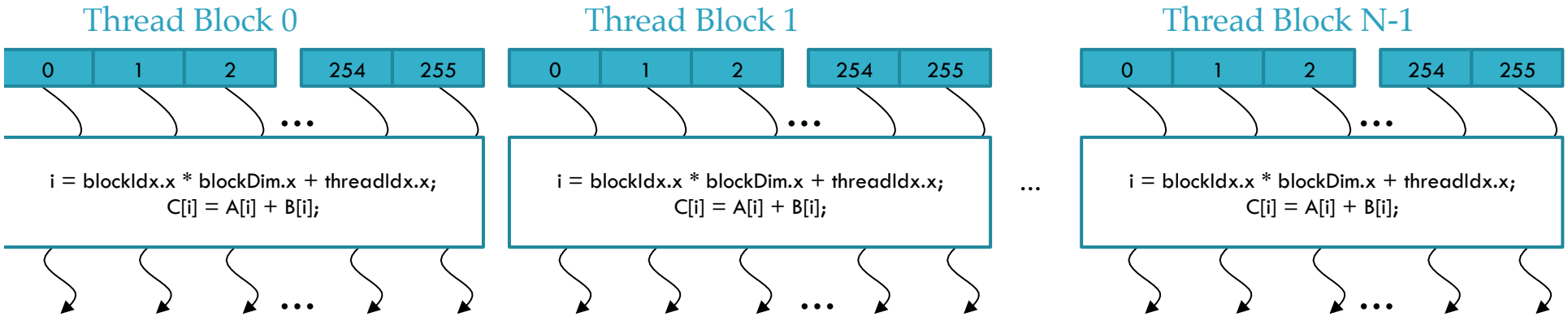


Arrays of Parallel Threads

- A CUDA kernel is executed by a grid (array) of threads
  - All threads in a grid run the same kernel code (Single Program Multiple Data)
  - Each thread has indexes that it uses to compute memory addresses and make control decisions



Thread Blocks: Scalable Cooperation

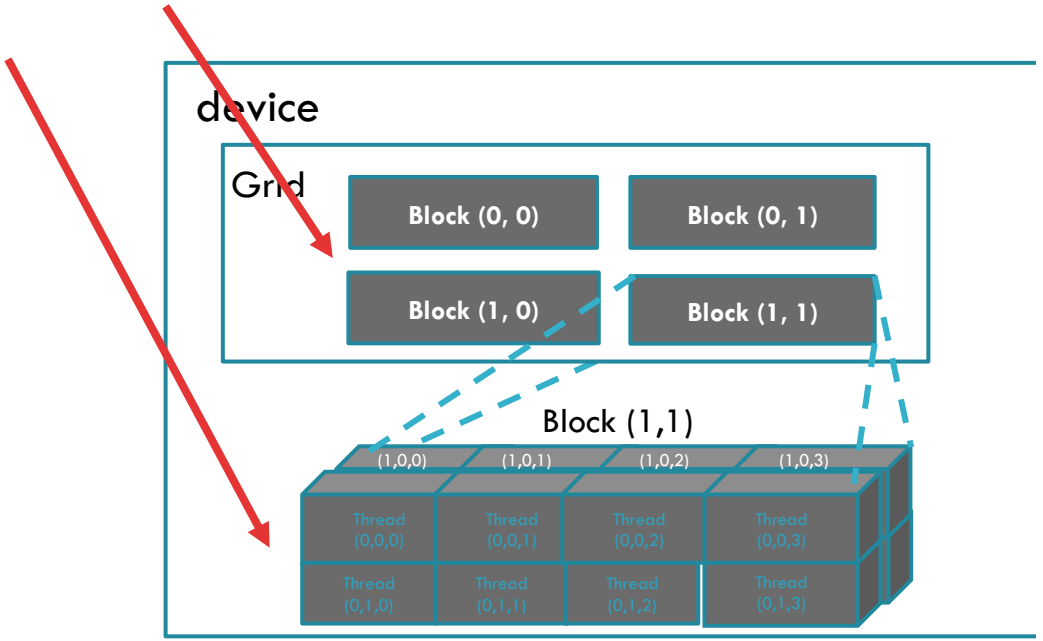


- Divide thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks do not interact

blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D, 2D, or 3D (CUDA 4.0)
  - threadIdx: 1D, 2D, or 3D

- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Handling arbitrary vector sizes

- Typical problems are not friendly multiples of blockDim.x, so we have to prevent accessing beyond the end of arrays:

```
// Add two vectors of size N: C[1..N] = A[1..N] + B[1..N]
__global__ void vecAdd(float* A, float* B, float* C, N) {
    int tid = threadIdx.x + (blockDim.x * blockIdx.x);
    if (tid < N)
      C[tid] = A[tid] + B[tid];
}
```

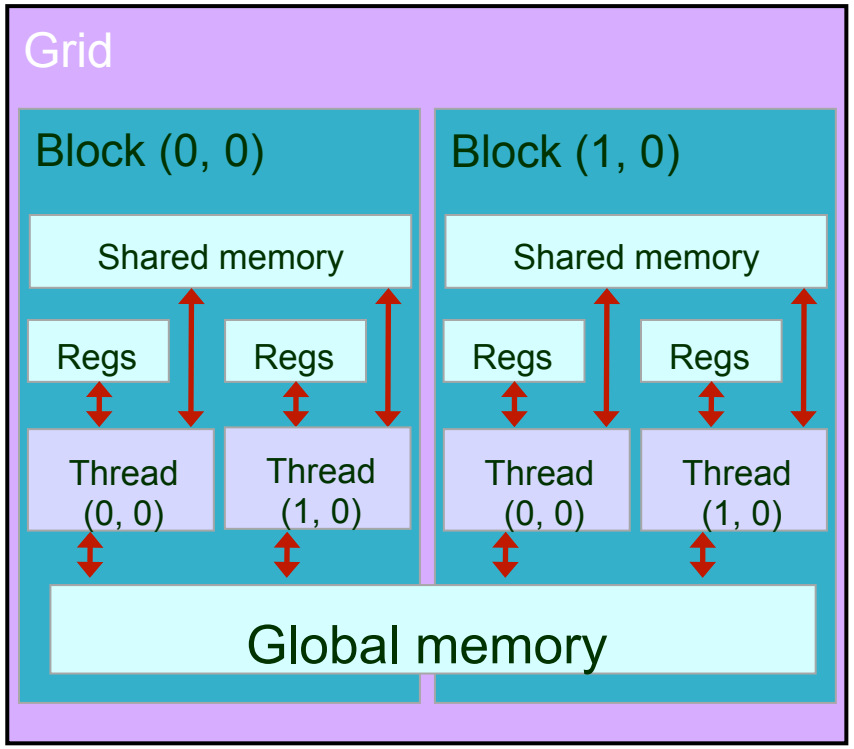
- And now, update the kernel launch to include the "incomplete" block of threads:

```
      vecAdd<<< (N + M-1)/256, 256>>>(d_A, d_B, d_C, N);
```

- Modifique seu trabalho para executar com um N arbitrário.

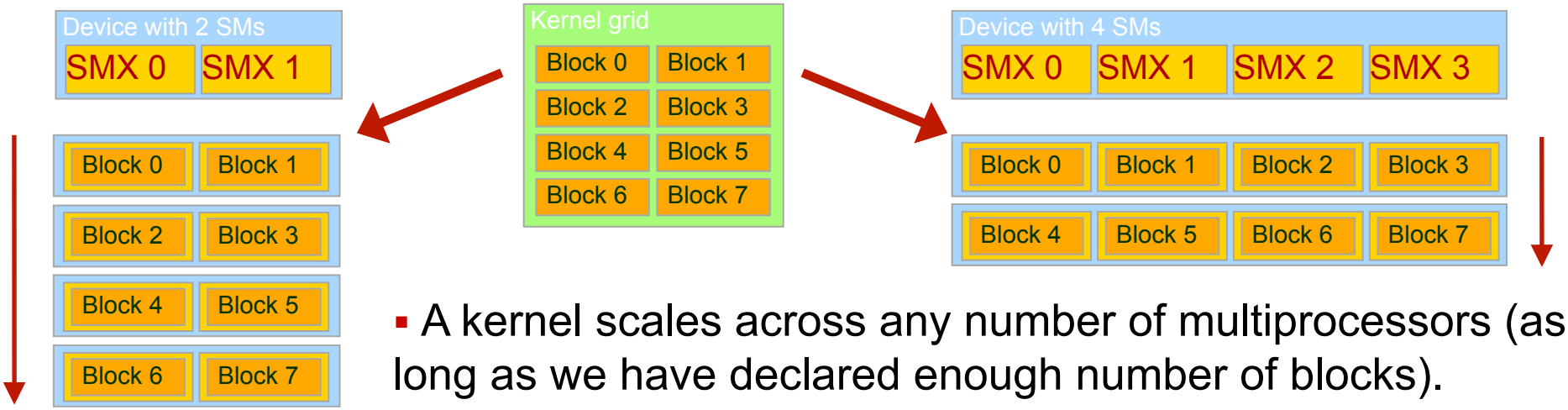
Summarizing about kernels, blocks, threads and parallelism

- Kernels are launched in grids.
- Each block executes fully on a single multiprocessor (SMX).
- Does not migrate.
- Several blocks can reside concurrently on one SMX.
- With control limitations. For example, in Kepler, we have:
  - Up to **16** concurrent blocks.
  - Up to **1024** threads per block.
  - Up to **2048** threads on each SMX.
- But usually, tighter limitations arise due to shared use of the register file and shared memory among all threads (as we have seen 3 slides ago).



Transparent scalability

- Since blocks cannot synchronize:
- The hardware is free to schedule the execution of a block on any multiprocessor.
- Blocks may run sequentially or concurrently depending on resources

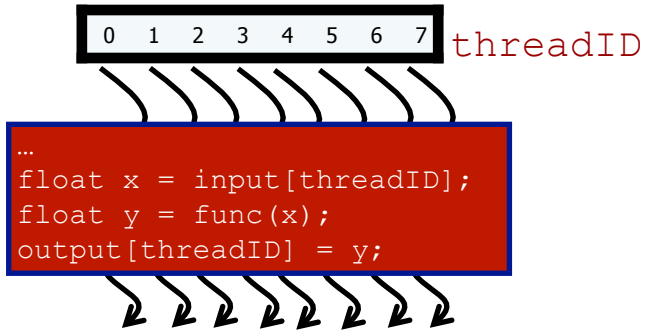


- A kernel scales across any number of multiprocessors (as long as we have declared enough number of blocks).

Kernels (and their relation with threads)

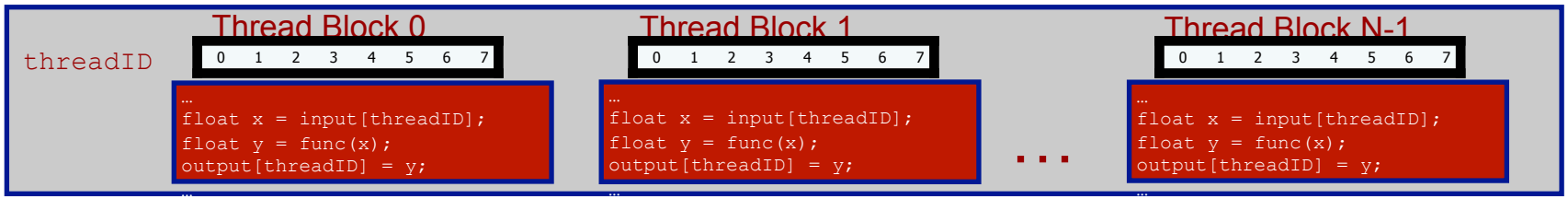
- Parallel portions of an application are executed on the device as kernels. All hardware resources and running threads are devoted to a single kernel, and when the kernel concludes, all GPU resources are freed for the next kernel (NO kernel multiprocessing). In other words:
  - Threads execute in parallel (within a block).
  - Blocks execute in parallel (within a kernel).
  - But kernels are launched sequentially.

- Depending on its threadID, each thread:
- Executes the same code on different data.
- Can make control decisions to distinguish from others.



Threads (and their relation with blocks)

- Thread cooperation is valuable:
- Share results to save computation.
- Share device memory accesses (for a drastic bandwidth reduction).
- The block guarantees performance and scalability, as it allows to replicate the thread-block execution as many times as required by the data volume:
- Enabling fine-grained parallelism.
- Without penalty, as context switch is free.



Questões que você deveria ser capaz de responder

1. Como executam os programas em uma máquina PRAM?
2. Como escrever programas que executam código na GPU?
3. Como compilar programas para executar na GPU?
4. O quê é uma arquitetura heterogênea?
5. Como descobrir as características da GPU em que vamos executar.

Redução paralela: algoritmo PRAM

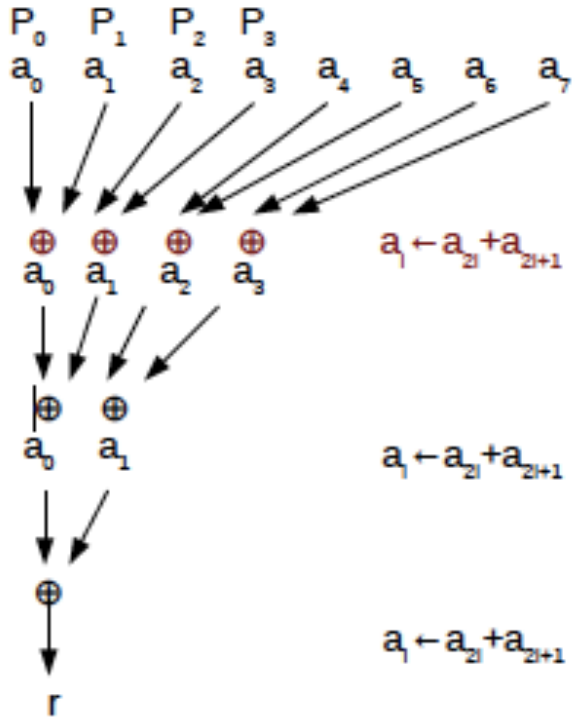
- Passo 1

Processador Pi:

a[i] = a[2*i] + a[2*i+1]

Inplace.

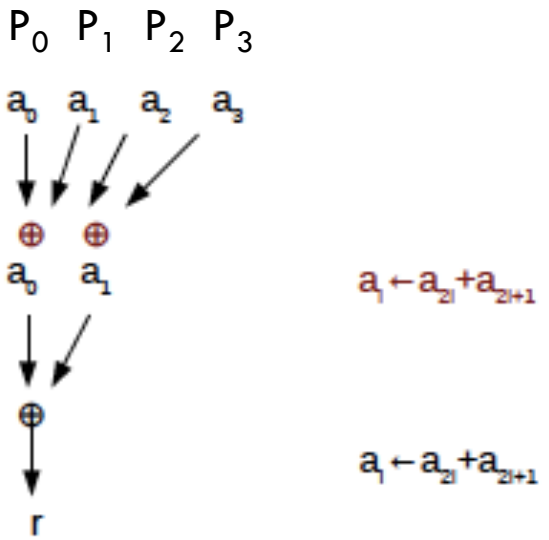
Funcionaria se não for síncrono?



Redução paralela: algoritmo PRAM

- Passo 2
  - Redução em n/2 processadores
  - Os outros processadores permanecen ociosos

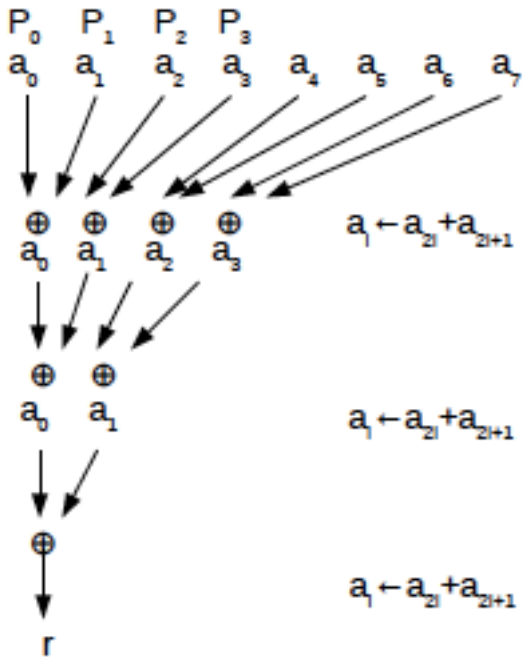
```
Processor Pi:
n'=n/2
if i < n' then
        a[i] = a[2*i] + a[2*i+1]
end if
```



Redução paralela: algoritmo PRAM

- Laço completo

```
Processor Pi:
   while n > 1 do
      if i < n then
         a[i] = a[2*i] + a[2*i+1]
      end if
     n = n / 2;
   end while
```



Complexity: O(log(n)) with O(n) processors

Interpretation of Brent's theorem

- We can design algorithms for a variable (unlimited) number of processors
  - They can run on any machine with fewer processors with known complexity

GPU Teaching Kit

Accelerated Computing

Lecture 2.4 – Introduction to CUDA C

Introduction to the CUDA Toolkit

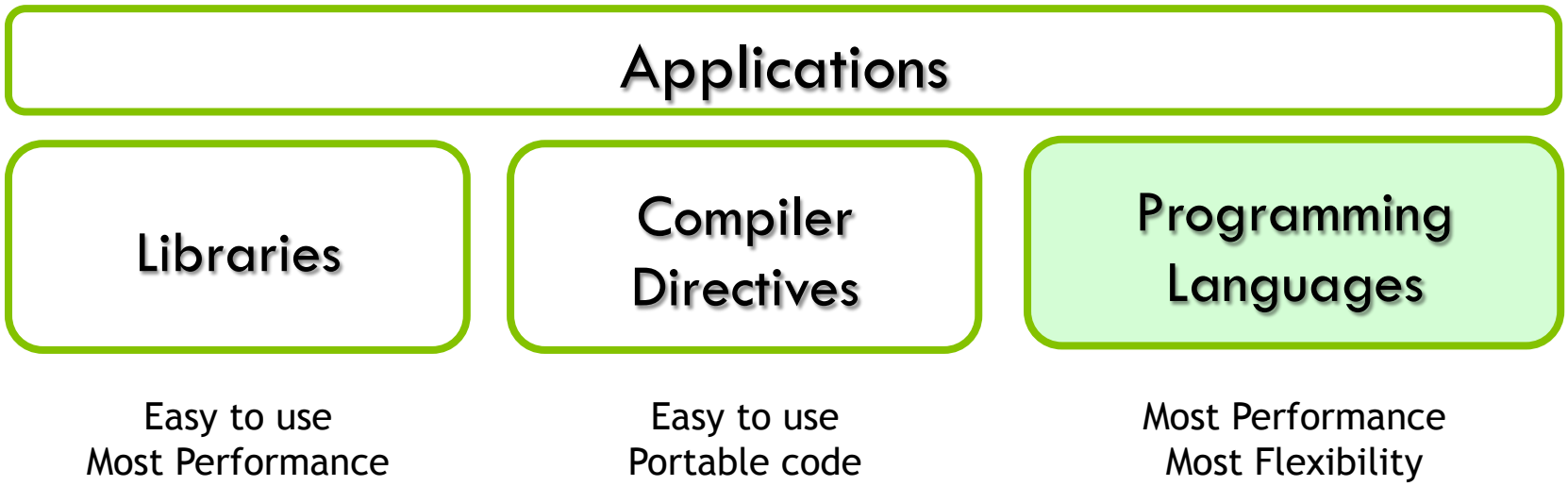
Objective

- To become familiar with some valuable tools and resources from the CUDA Toolkit
  - Compiler flags
  - Debuggers
  - Profilers

GPU Programming Languages

Numerical analytics	MATLAB, Mathematica, LabVIEW
Fortran	CUDA Fortran
C	CUDA C
C++	CUDA C++
Python	PyCUDA, Copperhead, Numba, NumbaPro
F#	Alea.cuBase

CUDA - C


Applications
Libraries
Compiler Directives
Programming Languages
Easy to use
Most Performance
Easy to use
Portable code
Most Performance
Most Flexibility

Developer Tools - Debuggers

NSIGHT

CUDA-GDB

CUDA MEMCHECK

NVIDIA Provided

allinea DDT

TotalView®

3rd Party

https://developer.nvidia.com/debugging-solutions

Compiler Flags

- Remember there are two compilers being used
  - NVCC: Device code
  - Host Compiler: C/C++ code
- NVCC supports some host compiler flags
  - If flag is unsupported, use –Xcompiler to forward to host
    - e.g. –Xcompiler –fopenmp
- Debugging Flags
  - -g: Include host debugging symbols
  - -G: Include device debugging symbols
  - -lineinfo: Include line information with symbols

CUDA-MEMCHECK

- Memory debugging tool
  - No recompilation necessary

    %> cuda-memcheck ./exe
- Can detect the following errors
  - Memory leaks
  - Memory errors (OOB, misaligned access, illegal instruction, etc)
  - Race conditions
  - Illegal Barriers
  - Uninitialized Memory
- For line numbers use the following compiler flags:
  - -Xcompiler -rdynamic -lineinfo

http://docs.nvidia.com/cuda/cuda-memcheck

Example 2: CUDA-MEMCHECK

Instructions:

1. Build & Run Example 2
   Output should be the numbers 0-9
   Do you get the correct results?
2. Run with cuda-memcheck
   %> cuda-memcheck ./a.out
3. Add nvcc flags "–Xcompiler –rdynamic –lineinfo"
4. Rebuild & Run with cuda-memcheck
5. Fix the illegal write

http://docs.nvidia.com/cuda/cuda-memcheck

CUDA-GDB

- cuda-gdb is an extension of GDB
  - Provides seamless debugging of CUDA and CPU code
- Works on Linux and Macintosh
  - For a Windows debugger use NSIGHT Visual Studio Edition

http://docs.nvidia.com/cuda/cuda-gdb

Example 3: cuda-gdb

Instructions:

1. Run exercise 3 in cuda-gdb

   %> cuda-gdb --args ./a.out

2. Run a few cuda-gdb commands:

```
(cuda-gdb) b main                          //set break point at main
(cuda-gdb) r                                   //run application
(cuda-gdb) l                                   //print line context
(cuda-gdb) b foo                               //break at kernel foo
(cuda-gdb) c                                   //continue
(cuda-gdb) cuda thread                     //print current thread
(cuda-gdb) cuda thread 10    //switch to thread 10
(cuda-gdb) cuda block                      //print current block
(cuda-gdb) cuda block 1                    //switch to block 1
(cuda-gdb) d                                   //delete all break points
(cuda-gdb) set cuda memcheck on          //turn on cuda memcheck
(cuda-gdb) r                                   //run from the beginning
```

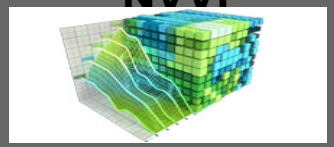
3. Fix Bug

http://docs.nvidia.com/cuda/cuda-gdb

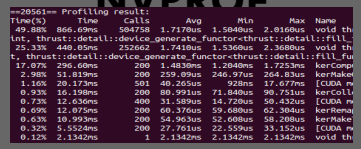
Developer Tools - Profilers

NSIGHT

NVVP

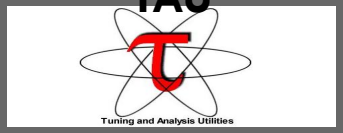


NVPROF



NVIDIA Provided

TAU



VampirTrace

3rd Party

https://developer.nvidia.com/performance-analysis-tools

NVPROF

Command Line Profiler

- Compute time in each kernel
- Compute memory transfer time
- Collect metrics and events
- Support complex process hierarchy's
- Collect profiles for NVIDIA Visual Profiler
- No need to recompile

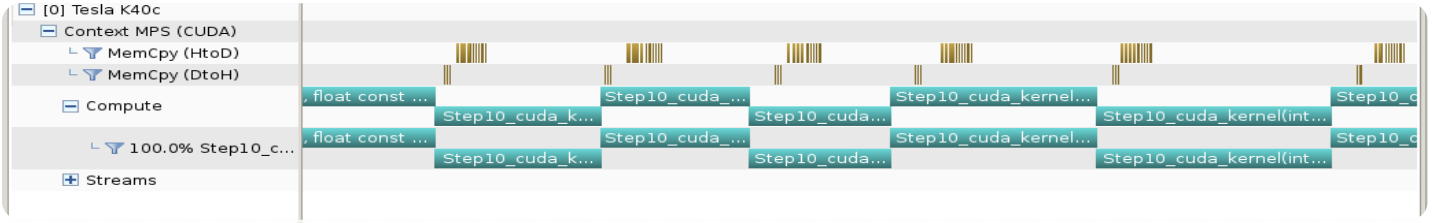
Example 4: nvprof

Instructions:

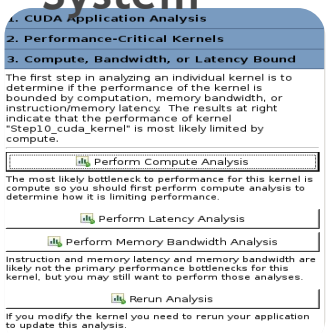
1. Collect profile information for the matrix add example
   %> nvprof ./a.out
2. How much faster is add_v2 than add_v1?
3. View available metrics
   %> nvprof --query-metrics
4. View global load/store efficiency
   %> nvprof --metrics gld_efficiency,gst_efficiency ./a.out
5. Store a timeline to load in NVVP
   %> nvprof –o profile.timeline ./a.out
6. Store analysis metrics to load in NVVP
   %> nvprof –o profile.metrics --analysis-metrics ./a.out

NVIDIA's Visual Profiler (NVVP)

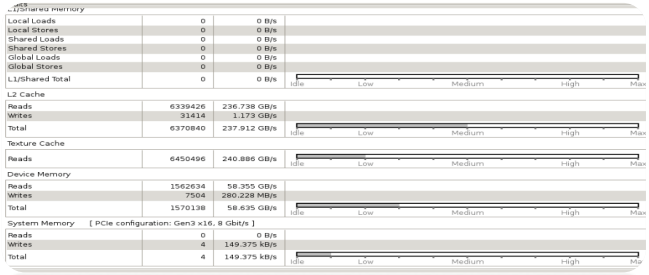
Timeline

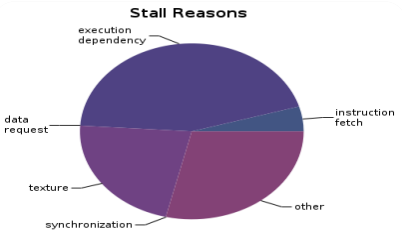


Guided System



Analysis





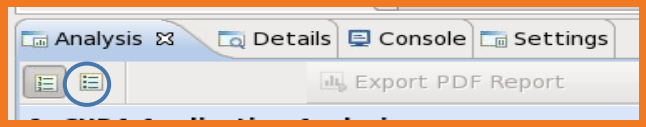
Example 4: NVVP

1. Import nvprof profile into NVVP
   - Launch nvvp
   - Click File/ Import/ Nvprof/ Next/ Single process/ Next / Browse
     - Select profile.timeline
   - Add Metrics to timeline
     - Click on 2nd Browse
     - Select profile.metrics
   - Click Finish
2. Explore Timeline
   - Control + mouse drag in timeline to zoom in
   - Control + mouse drag in measure bar (on top) to measure

Example 4: NVVP

Instructions:

1. Click on a kernel
2. On Analysis tab click on the unguided analysis



2. Click Analyze All

   Explore metrics and properties

   What differences do you see between the two kernels?

Note:

If kernel order is non-deterministic you can only load the timeline or the metrics but not both.

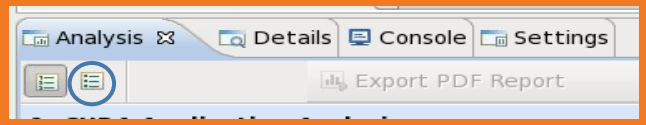
If you load just metrics the timeline looks odd but metrics are correct.

Example 4: NVVP

Let's now generate the same data within NVVP

Instructions:

1. Click File / New Session / Browse
   Select Example 4/a.out
   Click Next / Finish



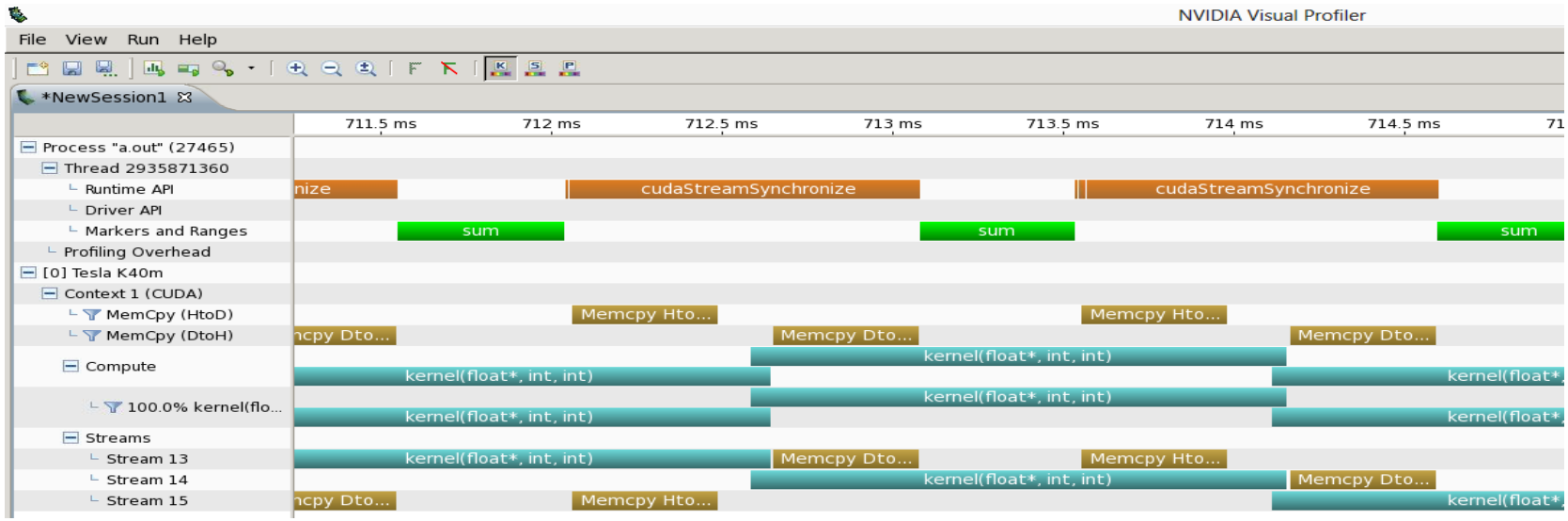
2. Click on a kernel
   Select Unguided Analysis
   Click Analyze All

NVTX

- Our current tools only profile API calls on the host
  - What if we want to understand better what the host is doing?
- The NVTX library allows us to annotate profiles with ranges
  - Add: #include <nvToolsExt.h>
  - Link with: -lnvToolsExt
- Mark the start of a range
  - nvtxRangePushA("description");
- Mark the end of a range
  - nvtxRangePop();
- Ranges are allowed to overlap

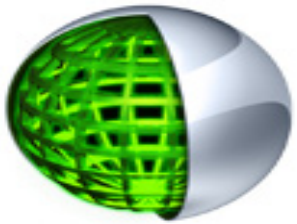
http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-generate-custom-application-profile-timelines-nvtx/

NVTX Profile



NSIGHT

- CUDA enabled Integrated Development Environment
  - Source code editor: syntax highlighting, code refactoring, etc
  - Build Manger
  - Visual Debugger
  - Visual Profiler
- Linux/Macintosh
  - Editor = Eclipse
  - Debugger = cuda-gdb with a visual wrapper
  - Profiler = NVVP
- Windows
  - Integrates directly into Visual Studio
  - Profiler is NSIGHT VSE



Example 4: NSIGHT

Let's import an existing Makefile project into NSIGHT

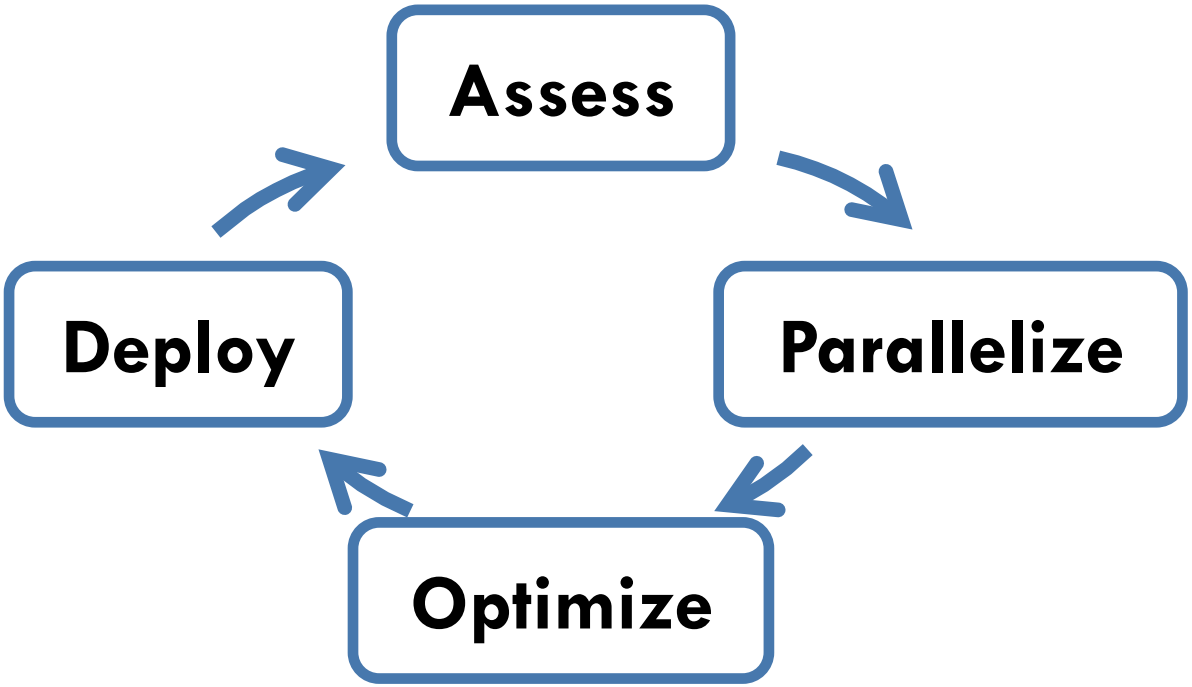
Instructions:

1. Run nsight
   Select default workspace
2. Click File / New / Makefile Project With Existing CodeTest
3. Enter Project Name and select the Example15 directory
4. Click Finish
5. Right Click On Project / Properties / Run Settings / New / C++ Application
6. Browse for Example 4/a.out
7. In Project Explorer double click on main.cu and explore source
8. Click on the build icon
9. Click on the run icon
10. Click on the profile icon

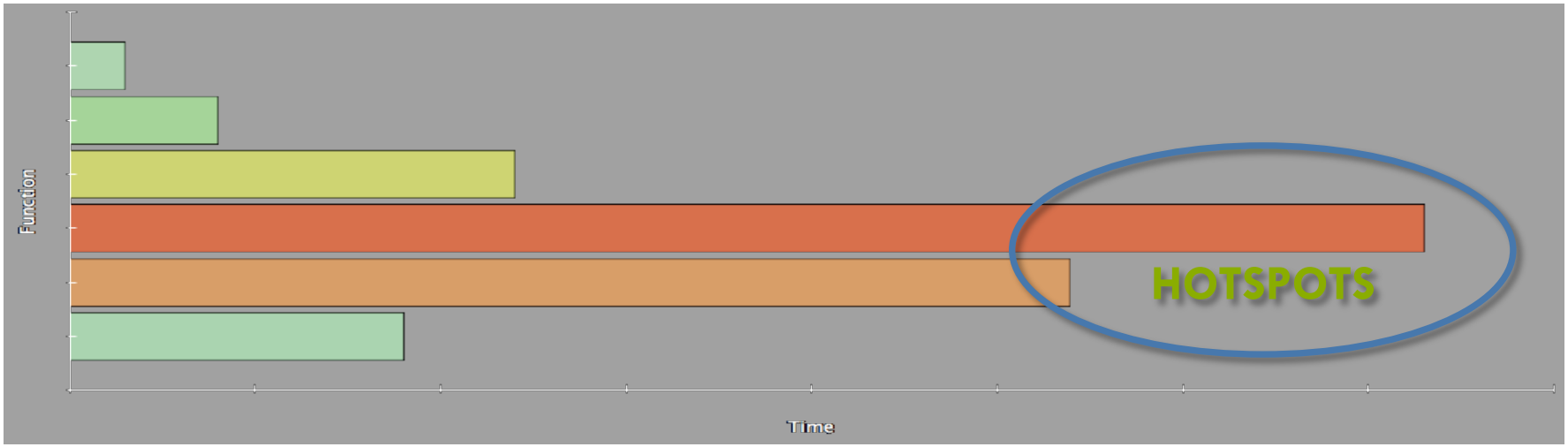
Profiler Summary

- Many profile tools are available
- NVIDIA Provided
  - NVPROF: Command Line
  - NVVP: Visual profiler
  - NSIGHT: IDE (Visual Studio and Eclipse)
- 3rd Party
  - TAU
  - VAMPIR

Optimization

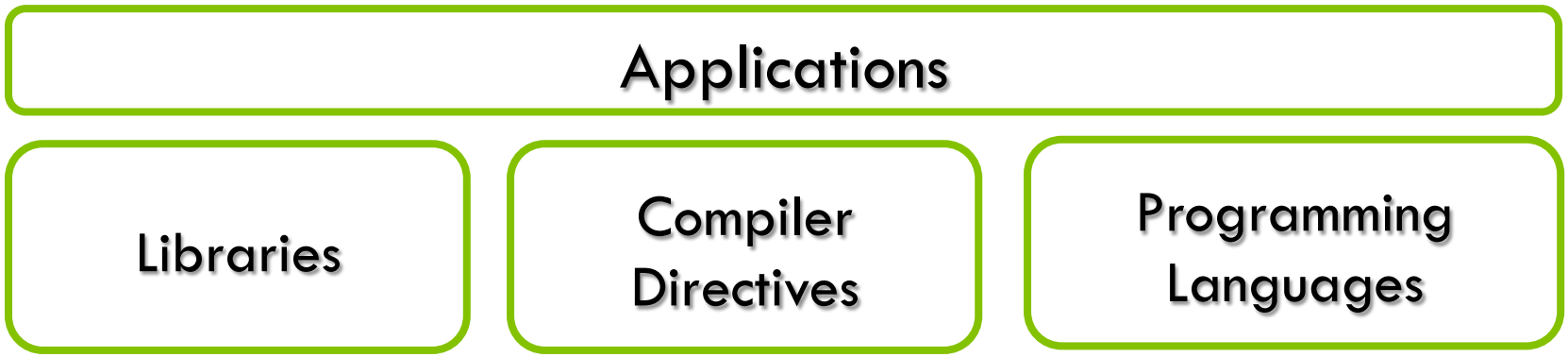


Assess



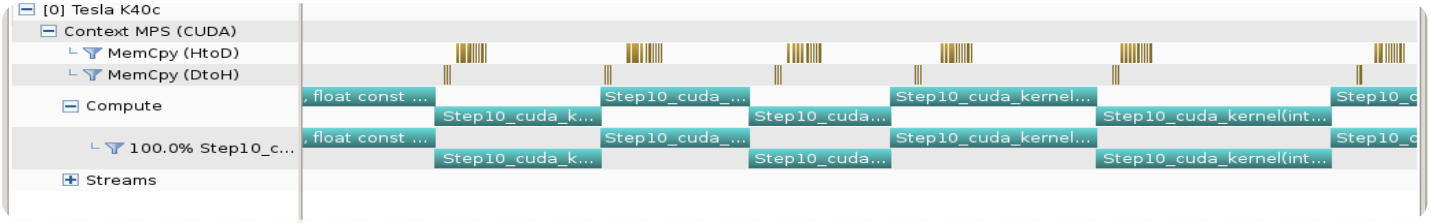
- Profile the code, find the hotspot(s)
- Focus your attention where it will give the most benefit

Parallelize

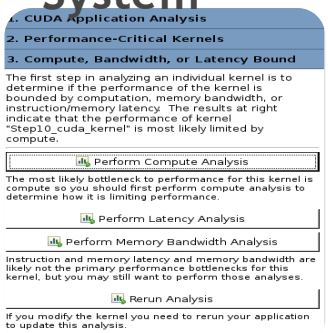


Optimize

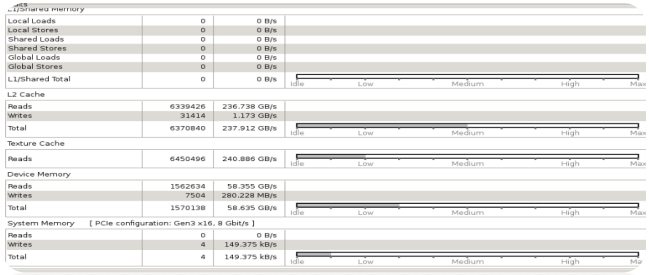
Timeline

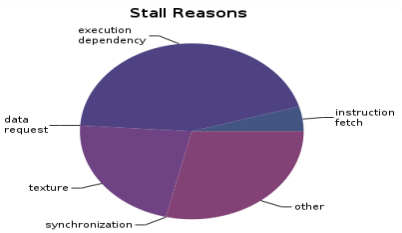


Guided System Analysis



Analysis





Bottleneck Analysis

- Don't assume an optimization was wrong
- Verify if it was wrong with the profiler

L1/Shared Memory

Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	2097152	1,351.979 GB/s	
Shared Stores	131072	84.499 GB/s	
Global Loads	131072	42.249 GB/s	
Global Stores	131072	42.249 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	2490368	1,520.977 GB/s	Idle Low Medium

gpuTranspose_kernel(int, int, float const *, float*)

Start	547.303 ms (5
End	547.716 ms (5
Duration	413.872 µs
Grid Size	[64,64,1]
Block Size	[32,32,1]
Registers/Thread	10
Shared Memory/Block	4 KiB
▾ Efficiency	
Global Load Efficiency	100%
Global Store Efficiency	100%
Shared Efficiency	⚠ 5.9%
Warp Execution Efficiency	100%
Non-Predicated Warp Execution Efficien	97.1%
▾ Occupancy	
Achieved	86.7%
Theoretical	100%
▾ Shared Memory Configuration	
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB

⚠ **Shared Memory Alignment and Access Pattern**

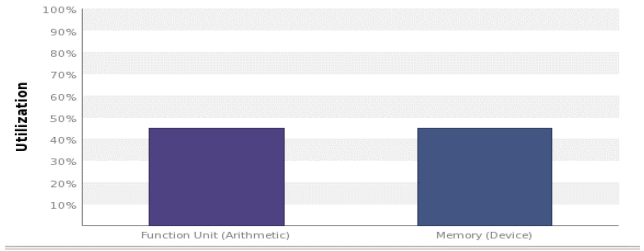
Memory bandwidth is used most efficiently when each shared memory load and store has proper alignment and access pattern.

Optimization: Select each entry below to open the source code to a shared load or store within the kernel with an inefficient alignment or access pattern. For each access pattern of the memory access.

▾ Line / File	main.cu - /home/jluitjens/code/CudaHandsOn/Example19
49	Shared Load Transactions/Access = 16, Ideal Transactions/Access = 1 [2097152 transactions for 131072 total executions]

Performance Analysis

gpuTranspose_kernel(int, int, float const *, float*
Start 770.067
End 770.324
Duration 256.714
Grid Size [64,64,1
Block Size [32,32,1
Registers/Thread 10
Shared Memory/Block 4.125 KiB
▽ Efficiency
Global Load Efficiency 100%
Global Store Efficiency 100%
Shared Efficiency ⚠ 50%
Warp Execution Efficiency 100%
Non-Predicated Warp Execution Efficien 97.1%
▽ Occupancy
Achieved 87.7%
Theoretical 100%
▽ Shared Memory Configuration
Shared Memory Requested 48 KiB
Shared Memory Executed 48 KiB



L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	131072	138.433 GB/s	
Shared Stores	131720	139.118 GB/s	
Global Loads	131072	69.217 GB/s	
Global Stores	131072	69.217 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	524936	415.984 GB/s	Idle Low Medium
L2 Cache			
L1 Reads	524288	69.217 GB/s	
L1 Writes	524288	69.217 GB/s	
Texture Reads	0	0 B/s	
Atomic	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Total	1048576	138.433 GB/s	Idle Low Medium
Texture Cache			
Reads	0	0 B/s	Idle Low Medium
Device Memory			
Reads	524968	69.306 GB/s	
Writes	524289	69.217 GB/s	
Total	1049257	138.523 GB/s	Idle Low Medium

GPU Teaching Kit