Fundamentals of Accelerating Computing with CUDA C/C++

NVIDIA Deep Learning Institute

About Me

Michel Agustin Herrera Sanchez, MSc, CQF

Master in High Performance Computing, University of A Coruña, Spain

OpenACC Standard Certified HPC GPU Mentor

NVIDIA DLI Certified Instructor – Accelerated Computing

BNP Paribas, Risk Systems Developer

Berlin, Germany

Workshop Outline

Introduction (15 min)

Accelerating Applications with CUDA C/C++ (120min)

Break (15 min)

Managed Accelerated Application Memory with CUDA C/C++ (120min)

Break (15 min)

Asynchronous Steaming and Visual Profiling for Accelerated Applications with CUDA/C++ (120min)

Final Review (15 min)

CUDA C is essentially C/C++ with a few extensions that allow one to execute functions on the GPU using many threads in parallel.

https://developer.nvidia.com/blog/easy-introduction-cuda-c-and-c/

Parallel Architecture – NVIDIA Tesla V100 GPU

7.8 TFLOPS FP64
15.7 TFLOPS FP32
900GB/s Memory Bandwidth

84 SMs
5376 FP32 cores
5376 INT32 cores
2688 FP64 cores
672 Tensor cores

Each SM has:
- 64 FP32 cores
- 64 INT32 cores
- 32 FP64 cores
- 8 Tensor Cores
- 4 Texture units

Memory Size 16GB
L2 Cache Size 6144KB
Shared Memory Size/SM 96KB
Register File Size/SM 256KB

https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

Parallel Architecture – Volta V100 Streaming Multi Processor (SM)

GPU Computing

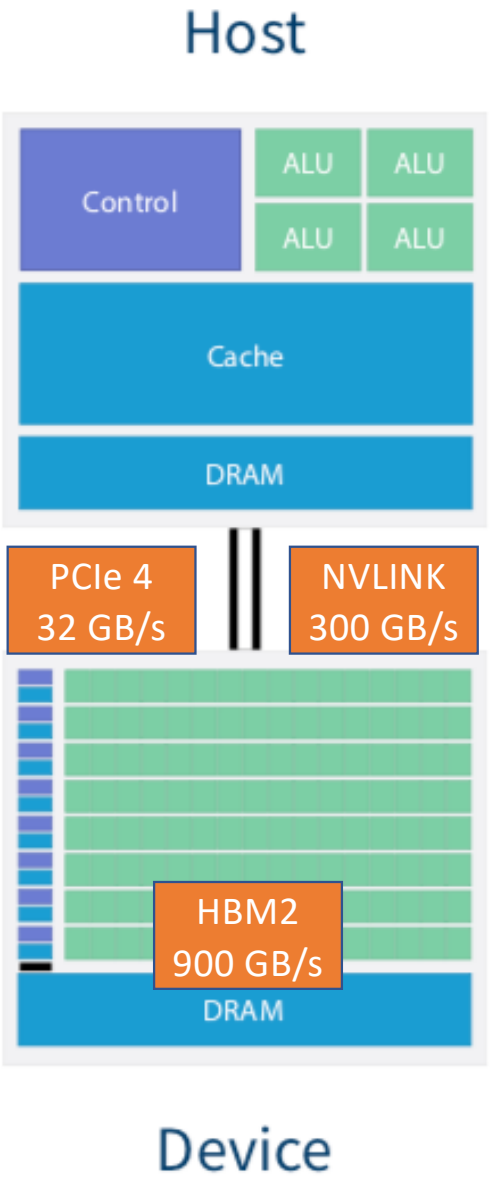
Differences between CPU and GPU Architectures

CPU

- More suitable for low latency workloads
- CPU has bigger memory caches
- Higher Processor Clock Rate
- Execution Model SIMD/SMT
- Data doesn't need to be moved from Host to Device to be processed

GPU

- More suitable for high throughput workloads
- GPU has less memory capacity but higher main memory bandwidth
- Lower Processor Clock Rate
- Execution Model SIMT
- Data Transfer between host and device is a Limiting performance factor



https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

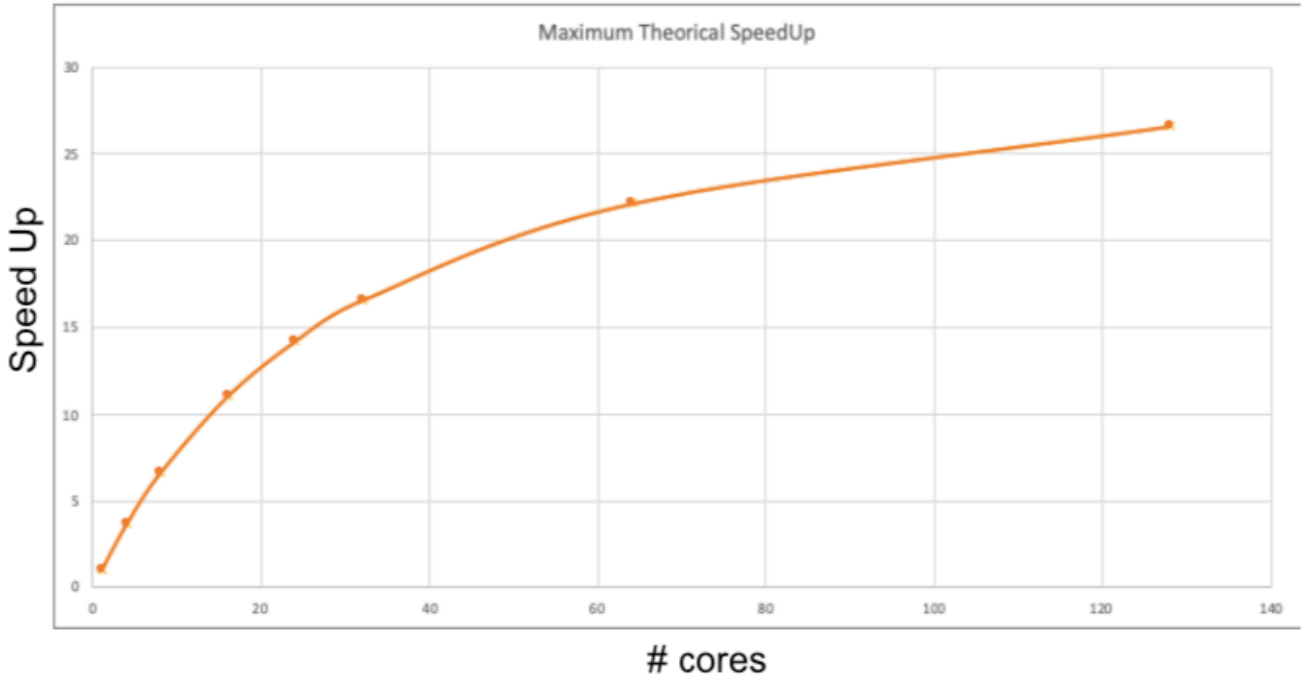
Andreas Herten 2022, JSC, CUDA Introduction PATC GPU Programming Course

Theorical Speed Up using Amdahls Law

Time(%)	Total Time (ns)	Instances	Average (ns)	Minimum (ns)	Maximum (ns)	StdDev (ns)	NVTX
33.9	18,600,340,372	1	18,600,340,372.0	18,600,340,372	18,600,340,372	0.0	total
33.1	18,136,889,464	1	18,136,889,464.0	18,136,889,464	18,136,889,464	0.0	main_simulation_loop
20.6	11,304,100,596	5,000	2,260,820.1	2,175,354	9,488,915	137,055.8	jacobi
12.4	6,822,128,866	5,000	1,364,425.8	1,338,596	6,271,088	140,306.2	swap
0.0	8,352,028	1	8,352,028.0	8,352,028	8,352,028	0.0	Init
0.0	1,970,085	5,000	394.0	136	746,645	10,559.8	calculate_error

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

# cores [n]	Theorical SpeedUp [S(n)]
1	1
4	3.669724771
8	6.611570248
16	11.03448276
24	14.20118343
32	16.58031088
64	22.14532872
128	26.61122661
Parallizable Fraction[P]	0.97



Application Performance
Memory Bound vs Compute Bound

- Compute to global memory access ratio (Arithmetic Intensity) is defined as the number of FLOPs performed for each byte access from the global memory within a kernel.

- Memory Bound Performance
  - Increase Speed up by making an efficient/reducing memory access

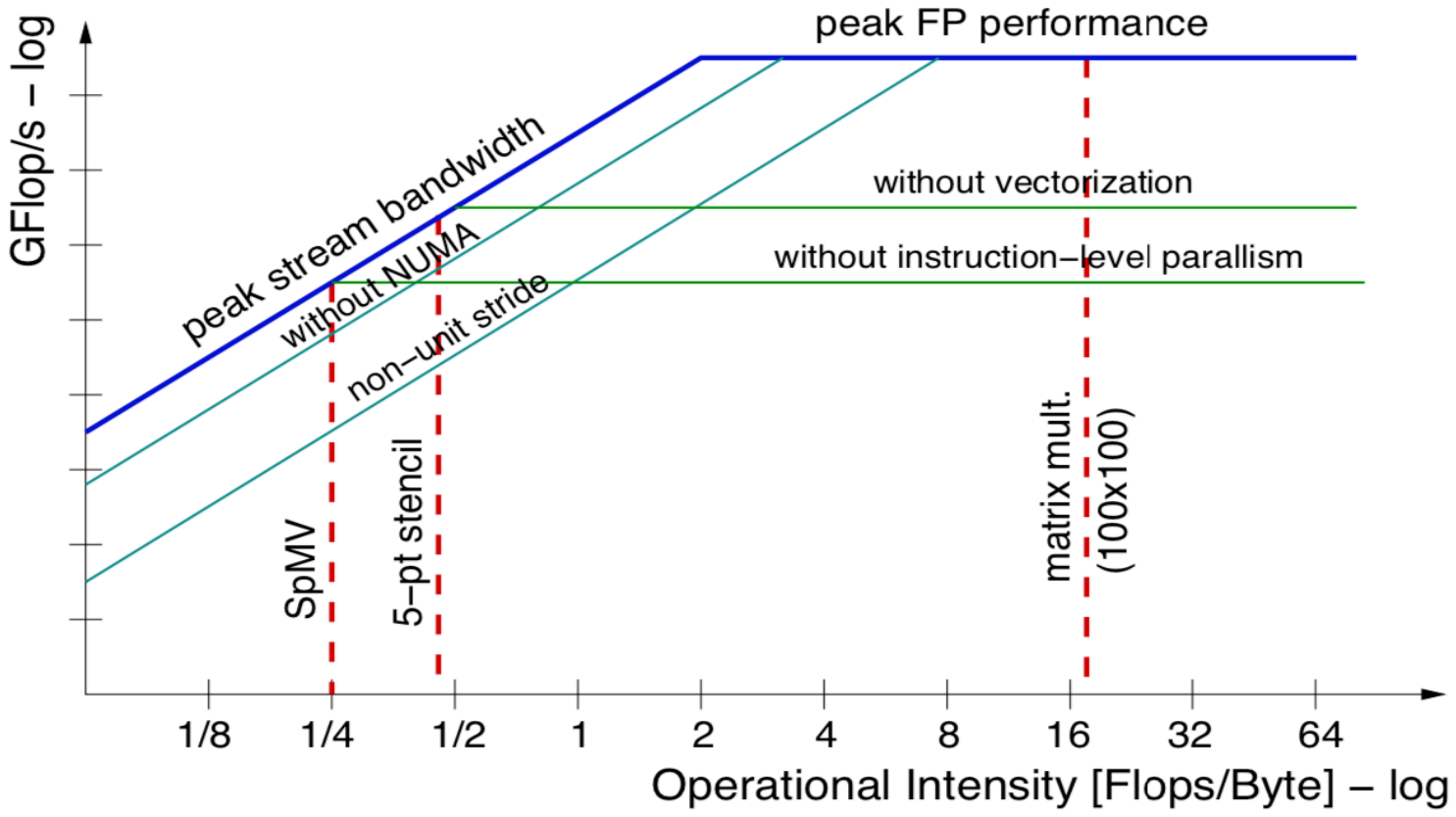
$$\frac{\#ops}{\#bytes} < \frac{BW_{math}}{BW_{mem}}$$

- Compute Bound Performance
  - Increase Speed up by reducing/optimizing compute operations

$$\frac{\#ops}{\#bytes} > \frac{BW_{math}}{BW_{mem}}$$

Wen Mei Hwu, David B Kirk Programming Massively Parallel Processors A Hands-on Approach 4th 2022
http://alexminnaar.com/2020/04/11/dl-gpu-perf-memory-vs-math.html

The Roof Line Model



Williams, Waterman, Patterson 2008

Programming GPUs with CUDA C/C++

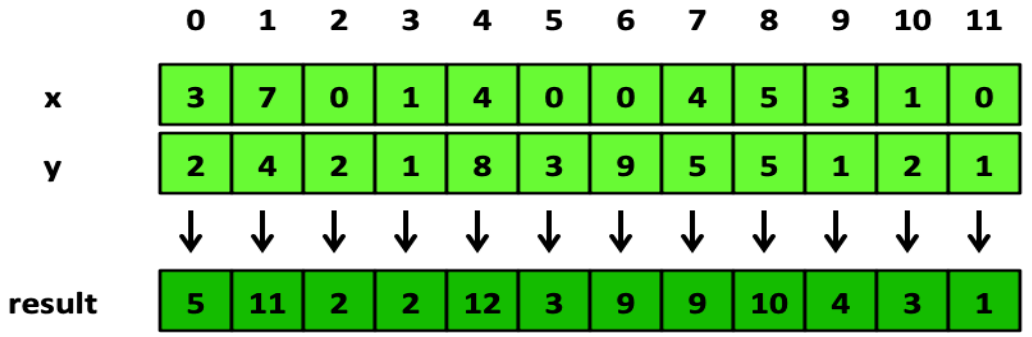
Programming GPUs with CUDA C/C++

```
#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y) {
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}

int main(void) {
      int N = 1<<20; // 1M elements
      float *x = (float*) malloc( N*sizeof(float) );
      float *y = (float*) malloc( N*sizeof(float) );

       // Initialize x, y  data
      for (int i = 0; i < N; i++) {
          x[i] = 1.0f; y[i] = 2.0f;
     }

      // Run kernel on 1M elements on the CPU
      add(N, x, y);

      // Check for errors (all values should be 3.0f)
      float maxError = 0.0f;

      for (int i = 0; i < N; i++)
          maxError = fmax(maxError, fabs(y[i]-3.0f));

      std::cout << "Max error: " << maxError << std::endl;

      // Free memory
      free(x);
      free(y);

      return 0;
}
```

https://developer.nvidia.com/blog/even-easier-introduction-cuda/

“add” CUDA Kernel (Monolithic) implementation

```
void add(int n, float *x, float *y) {

      for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}
```

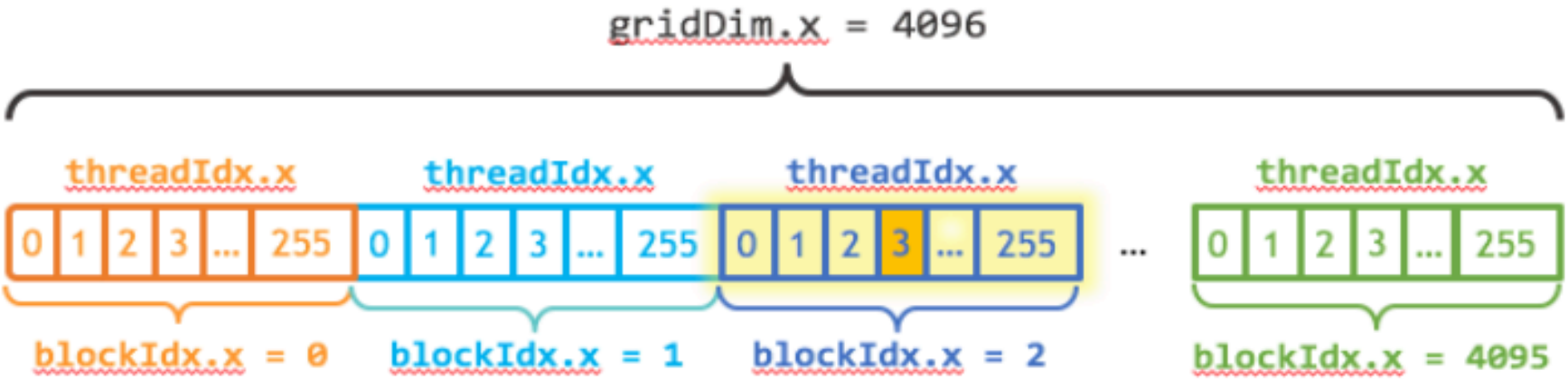


Parallelize a loop using a kernel.
Launch One Thread Per Data Element

```
__global__ void add(int n, float *x, float *y) {

      int index = blockIdx.x * blockDim.x + threadIdx.x;

      if (index < n) {
        y[index] = x[index] + y[index];
      }
}
```

https://ipcc.cs.uoregon.edu/lectures/lecture-5-patterns.pdf

Grid / Thread Blocks / Threads




```
index = blockIdx.x  * blockDim.x + threadIdx.x

index =    (2)      *   (256)    +     (3)      = 515
```

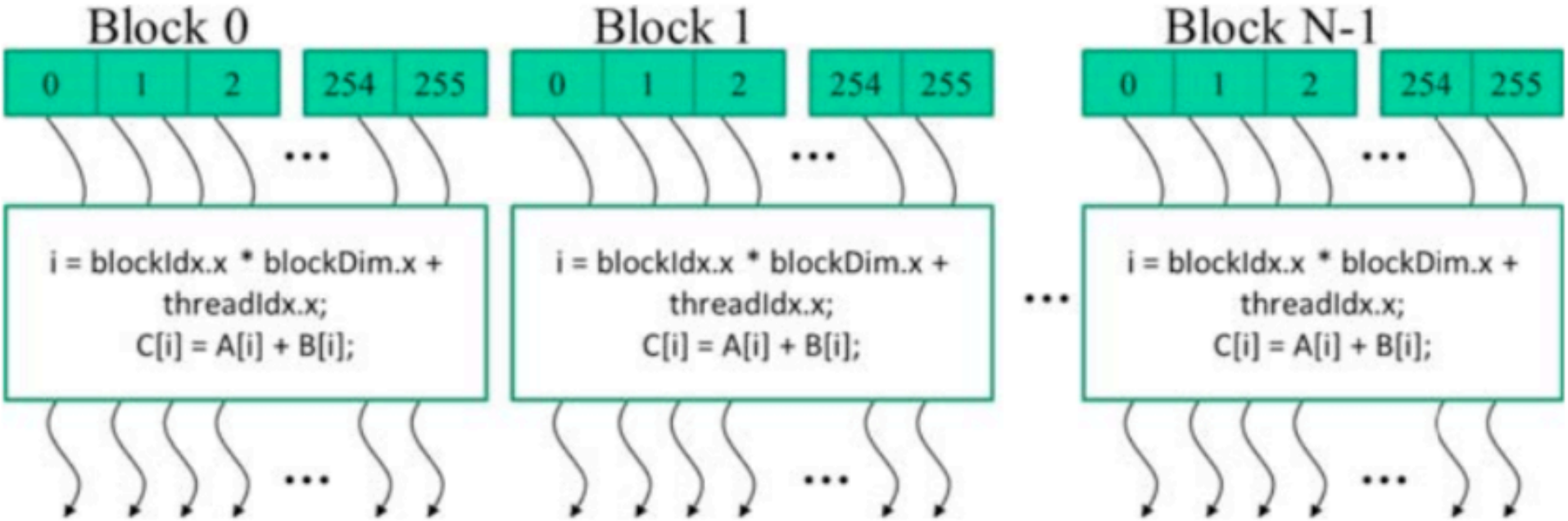
https://developer.nvidia.com/blog/even-easier-introduction-cuda/

CUDA Kernel Execution Configuration

```
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```

https://developer.nvidia.com/blog/even-easier-introduction-cuda/

Grid / Thread Blocks / Threads



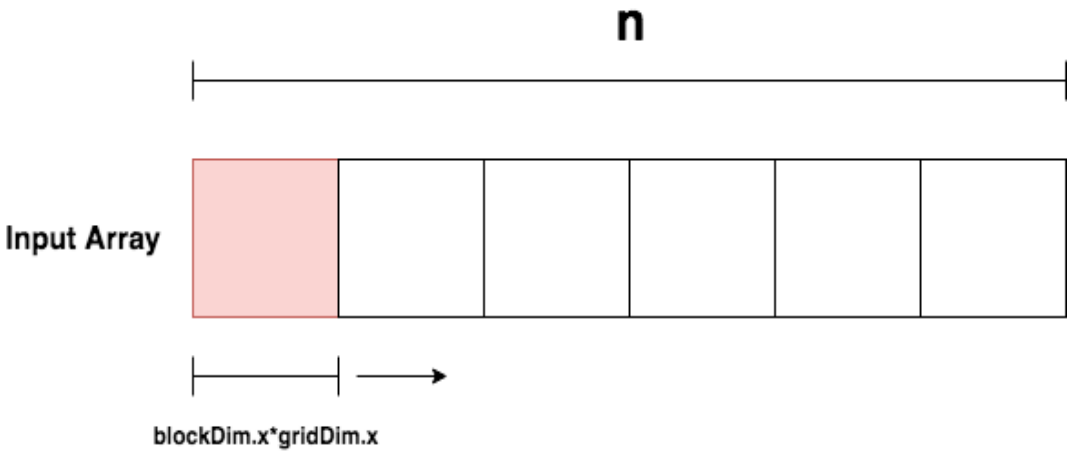
All Threads in a grid execute the same kernel

https://docs.nvidia.com/cuda/#programming-guides

Wen Mei Hwu, David B Kirk Programming Massively Parallel Processors A Hands-on Approach 4th 2022

“add” CUDA Kernel (Grid Stride Loop)

```
__global__ void add(int n, float *x, float *y) {

    int index = blockIdx.x * blockDim.x + threadIdx.x;

    int stride = blockDim.x * gridDim.x;

    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
}
```



https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/

https://alexminnaar.com/2019/08/02/grid-stride-loops.html

https://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf

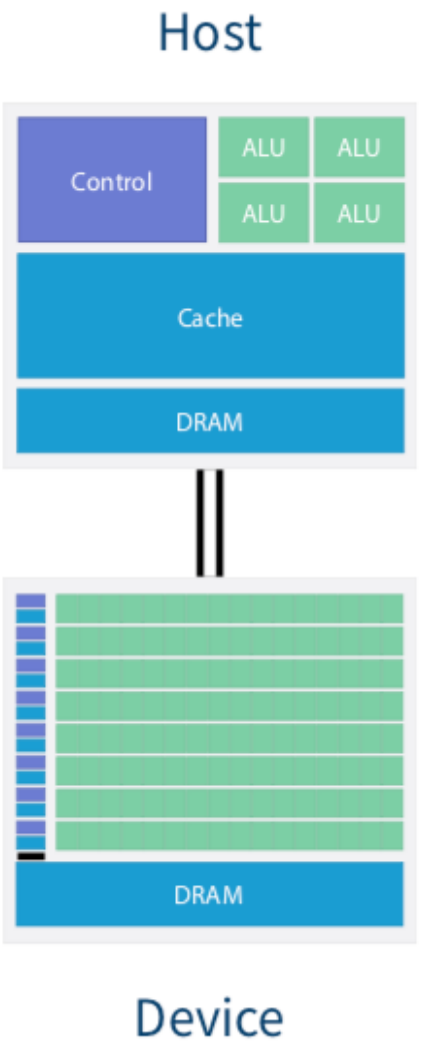
Grid Stride Loop Execution Configuration

```
int numSMs;
cudaDeviceGetAttribute(&numSMs, cudaDevAttrMultiProcessorCount, 0);

int blockSize = 256;
int numBlocks = 32 * numSMs;
add<<<numBlocks, blockSize>>>(N, x, y);
```

https://developer.nvidia.com/blog/even-easier-introduction-cuda/

Host/Device Memory Model

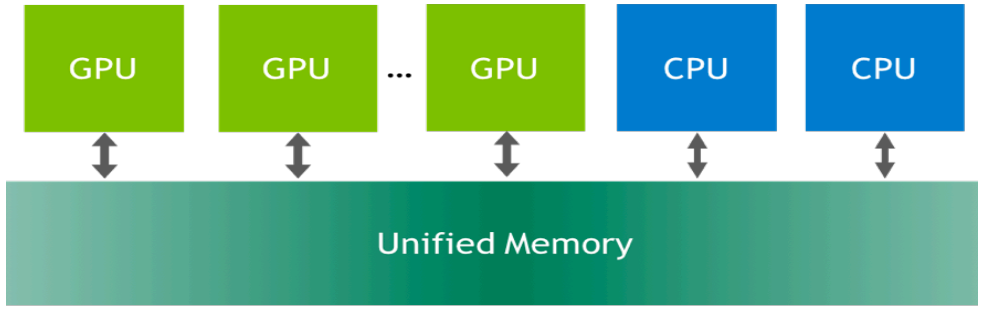
```
float *x, *y, *x_d, *y_d;

int size = N*sizeof(float));

// Allocate Host Memory
x = (float*) malloc( size);
y = (float*) malloc( size);

// Allocate Device Memory
cudaMalloc((void**)&x_d, size);
cudaMalloc((void**)&y_d, size);

// Initialize x, y data arrays ….

// Copy Data from Host to Device
cudaMemcpy(x_d, x, size, cudaMemcpyHostToDevice);
cudaMemcpy(y_d, y, size, cudaMemcpyHostToDevice);

// Execute CUDA Kernel ….
// …

// Copy Data from Device to Host
cudaMemcpy(y, y_d, size, cudaMemcpyDeviceToHost);

// Free memory
cudaFree(x_d); cudaFree(y_d);
free(x); free(y);
```



https://developer.nvidia.com/blog/even-easier-introduction-cuda/

CUDA Unified Memory Model

```
// Allocate Memory accessible from CPU or GPU

float *x, *y;
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));

 ...
// Free memory
cudaFree(x);
cudaFree(y);
```



https://developer.nvidia.com/blog/even-easier-introduction-cuda/

Programming GPUs with CUDA C/C++

```
#include <iostream>
#include <math.h>

// C CUDA Kernel
__global__ void add(int n, float *x, float *y) { //…}

int main(void) {
      int N = 1<<20; // 1M elements
      float *x, *y, *x_d, *y_d;
      int size = N*sizeof(float) ;

      // Allocate Host Memory
      x = (float*) malloc( size);
      y = (float*) malloc( size);

      // Allocate Device Memory
       cudaMalloc((void**)&x_d, size);
       cudaMalloc((void**)&y_d, size);

       // Initialize data x, y on host ….

       // Copy Data from Host to Device
       cudaMemcpy(x_d, x, size, cudaMemcpyHostToDevice);
       cudaMemcpy(y_d, y, size, cudaMemcpyHostToDevice);

      // Run kernel on 1M elements on the GPU
      int blockSize = 256;
      int numBlocks = (N + blockSize - 1) / blockSize;
      add<<<numBlocks, blockSize>>>(N, x, y);

      // Copy Data from Device to Host
      cudaMemcpy(y, y_d, size, cudaMemcpyDeviceToHost);

      // Check for errors (all values should be 3.0f)
      float maxError = 0.0f;

      for (int i = 0; i < N; i++)
          maxError = fmax(maxError, fabs(y[i]-3.0f));

      std::cout << "Max error: " << maxError << std::endl;

      // Free memory
      cudaFree(x_d);   cudaFree(y_d);

      free(x); free(y);

      return 0;
}
```

Programming GPUs with CUDA C/C++

```
#include <iostream>
#include <math.h>

// C CUDA Kernel
__global__ void add(int n, float *x, float *y) { //…
}

int main(void) {
      int N = 1<<20; // 1M elements
      float *x, *y ;

       // Allocate Unified Memory
       cudaMallocManaged(&x, N*sizeof(float));
       cudaMallocManaged(&y, N*sizeof(float));

     // Initialize x, y  data on host
      for (int i = 0; i < N; i++)  {
          x[i] = 1.0f; y[i] = 2.0f;
     }
```

```
      // Run kernel on 1M elements on the GPU
      int blockSize = 256;
      int numBlocks = (N + blockSize - 1) / blockSize;
      add<<<numBlocks, blockSize>>>(N, x, y);

      // Wait for GPU to finish before accessing on host
      cudaDeviceSynchronize();

      // Check for errors (all values should be 3.0f)
      float maxError = 0.0f;

      for (int i = 0; i < N; i++)
          maxError = fmax(maxError, fabs(y[i]-3.0f));

      std::cout << "Max error: " << maxError << std::endl;

      // Free memory
      cudaFree(x);
      cudaFree(y);

      return 0;
}
```

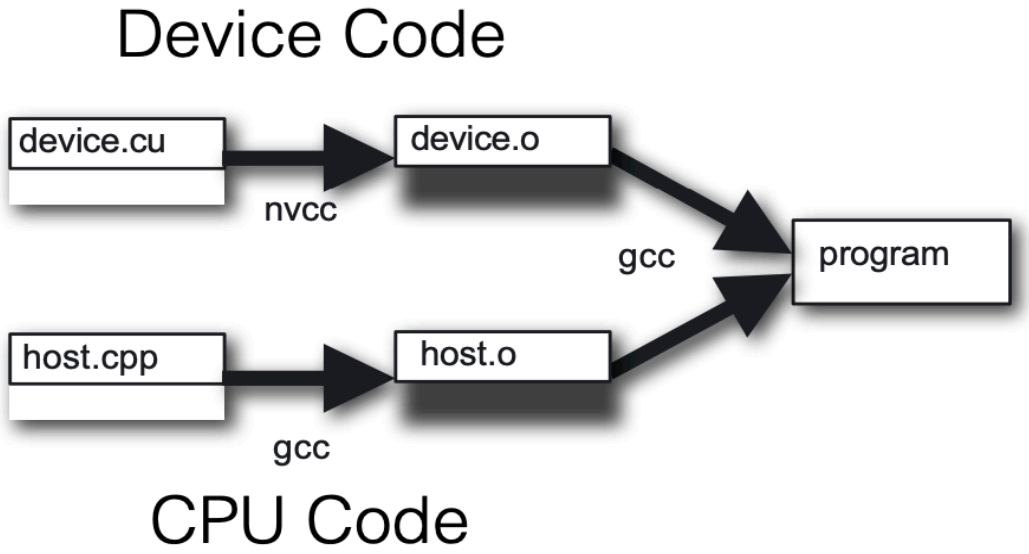
Compiling, Executing and Profiling CUDA C/C++ Kernels

GPU Compilation and Execution

> nvcc add.cu -o add_cuda

> ./add_cuda

Max error: 0.000000



https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#gpu-compilation

Profiling

> nsys profile --stats=true --force-overwrite=true -o nsys-report ./add_cuda

Max error: 0.000000

Time(%)	Total Time (ns)	Instances	Average (ns)	Minimum (ns)	Maximum (ns)	StdDev (ns)	NVTX
38.4	3,282,267,380	1	3,282,267,380.0	3,282,267,380	3,282,267,380	0.0	total
33.0	2,824,520,551	1	2,824,520,551.0	2,824,520,551	2,824,520,551	0.0	main_simulation_loop
14.3	1,223,043,756	5,000	244,608.8	227,902	1,236,437	38,328.4	jacobi
14.2	1,210,402,922	5,000	242,080.6	226,149	477,625	33,842.8	swap
0.1	7,693,156	1	7,693,156.0	7,693,156	7,693,156	0.0	init
0.0	1,636,038	5,000	327.2	136	547,247	7,740.7	calculate_error

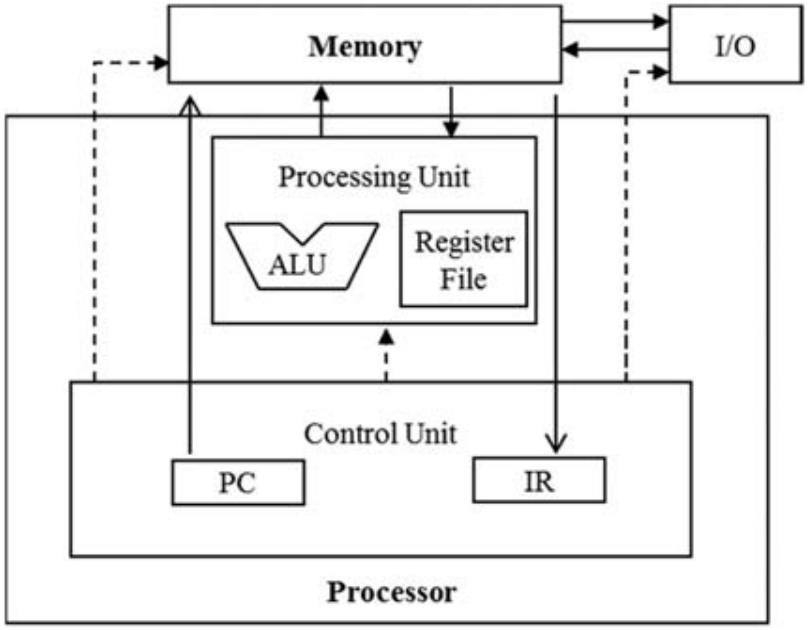
https://developer.nvidia.com/blog/cuda-pro-tip-generate-custom-application-profile-timelines-nvtx/
https://docs.nvidia.com/nsight-systems/UserGuide/index.html

Performance Summary

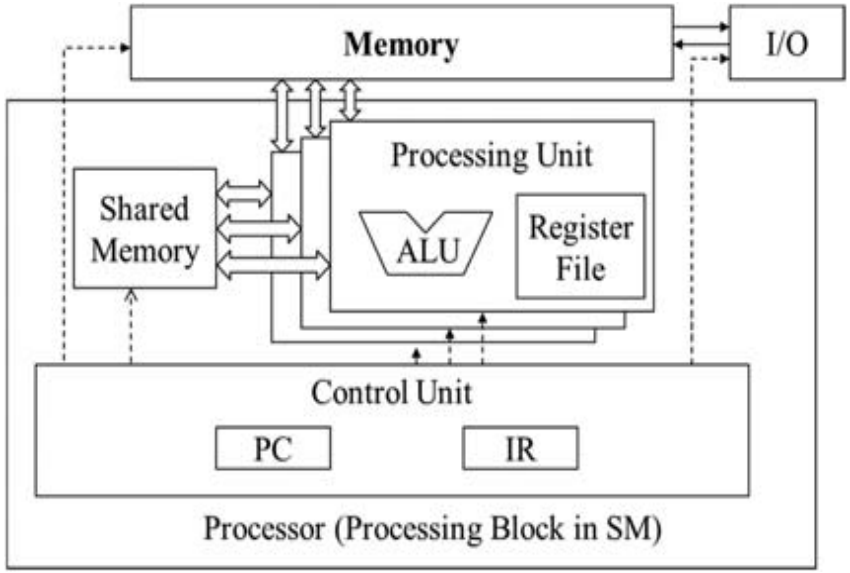
	Laptop (GeForce GT 750M)		Server (Tesla K80)	
Version	Time	Bandwidth	Time	Bandwidth
1 CUDA Thread	411ms	30.6 MB/s	463ms	27.2 MB/s
1 CUDA Block	3.2ms	3.9 GB/s	2.7ms	4.7 GB/s
Many CUDA Blocks	0.68ms	18.5 GB/s	0.094ms	134 GB/s

https://developer.nvidia.com/blog/even-easier-introduction-cuda/

CPU



GPU / Shared Memory

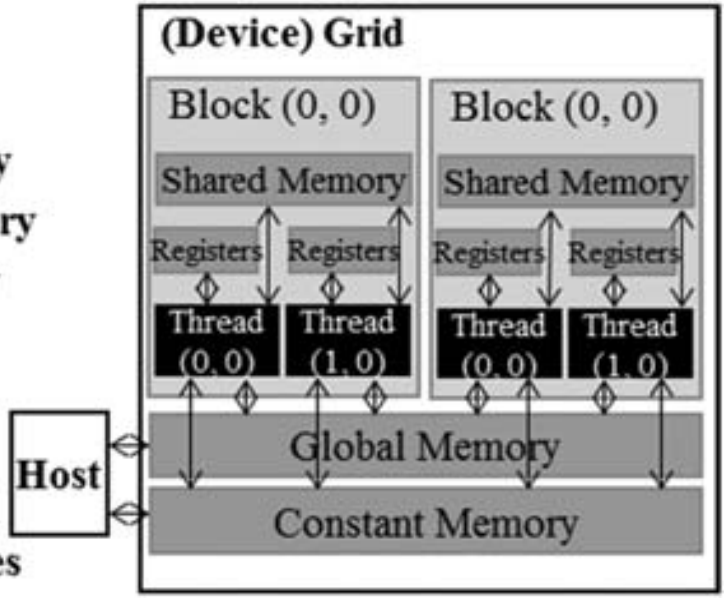


https://www.olcf.ornl.gov/cuda-training-series/
https://www.olcf.ornl.gov/wp-content/uploads/2019/12/02-CUDA-Shared-Memory.pdf
Wen Mei Hwu, David B Kirk Programming Massively Parallel Processors A Hands-on Approach 4th 2022

Overview of CUDA Device Memory Model

Device code can:
- R/W per-thread **registers**
- R/W per-thread **local memory**
- R/W per-block **shared memory**
- R/W per-grid **global memory**
- Read only per-grid **constant memory**

Host code can
- Transfer data to/from per grid **global** and **constant memories**



Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Grid
Automatic array variables	Local	Thread	Grid
__device__ __shared__ int SharedVar;	Shared	Block	Grid
__device__ int GlobalVar;	Global	Grid	Application
__device__ __constant__ int ConstVar;	Constant	Grid	Application

Matrix Matrix Multiplication

Access order

$thread_{0,0}$	$M_{0,0} * N_{0,0}$	$M_{0,1} * N_{1,0}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
$thread_{0,1}$	$M_{0,0} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
$thread_{1,0}$	$M_{1,0} * N_{0,0}$	$M_{1,1} * N_{1,0}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
$thread_{1,1}$	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$

FIGURE 5.6

Global memory accesses performed by threads in $block_{0,0}$.

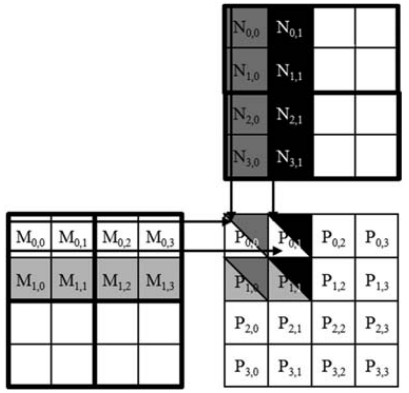


FIGURE 5.7

Tiling M and N to utilize shared memory.

	Phase 0			Phase 1		
$thread_{0,0}$	$\mathbf{M_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{0,1}*Nds_{1,0}$	$\mathbf{M_{0,2}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{2,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{0,1}*Nds_{1,0}$
$thread_{0,1}$	$\mathbf{M_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{0,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}*Nds_{0,1}$ + $Mds_{0,1}*Nds_{1,1}$	$\mathbf{M_{0,3}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{2,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}*Nds_{0,1}$ + $Mds_{0,1}*Nds_{1,1}$
$thread_{1,0}$	$\mathbf{M_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{1,0}$	$\mathbf{M_{1,2}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{3,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{1,0}$
$thread_{1,1}$	$\mathbf{M_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{1,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}*Nds_{0,1}$ + $Mds_{1,1}*Nds_{1,1}$	$\mathbf{M_{1,3}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{3,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}*Nds_{0,1}$ + $Mds_{1,1}*Nds_{1,1}$

time

FIGURE 5.8

Execution phases of a tiled matrix multiplication.

Wen Mei Hwu, David B Kirk Programming Massively Parallel Processors A Hands-on Approach 4th 2022

```
01    #define TILE_WIDTH 16
02    __global__ void matrixMulKernel(float* M, float* N, float* P, int Width) {
03
04        __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
05        __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
06
07        int bx = blockIdx.x;  int by = blockIdx.y;
08        int tx = threadIdx.x; int ty = threadIdx.y;
09
10        // Identify the row and column of the P element to work on
11        int Row = by * TILE_WIDTH + ty;
12        int Col = bx * TILE_WIDTH + tx;
13
14        // Loop over the M and N tiles required to compute P element
15        float Pvalue = 0;
16        for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
17
18            // Collaborative loading of M and N tiles into shared memory
19            Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
20            Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
21            __syncthreads();
22
23            for (int k = 0; k < TILE_WIDTH; ++k) {
24                Pvalue += Mds[ty][k] * Nds[k][tx];
25            }
26            __syncthreads();
27
28        }
29        P[Row*Width + Col] = Pvalue;
30
31    }
```

FIGURE 5.9

A tiled matrix multiplication kernel using shared memory.