Introduction to GPU programming using CUDA

Dr. Ezhilmathi Krishnasamy

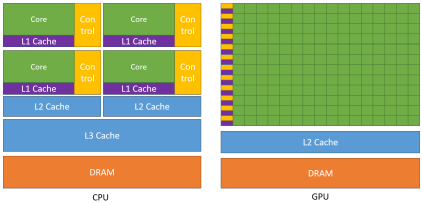
January 21^{st}, 2025

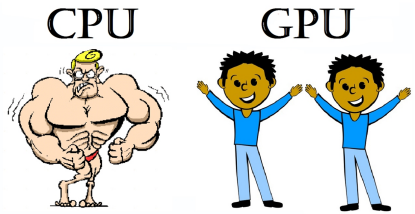
Outline

Outline for section 1

Important differences between CPU and GPU

- A GPU has many more cores compared to a CPU.
- On the other hand, the CPU's frequency is higher than that of the GPU, which makes the CPU faster in computing than the GPU.
  - Intel® Core™ i7-10700K Processor base frequency is **3.80 GHz**, whereas Nvidia Ampere has **0.765 GHz**.
  - The higher the frequency, the faster the processor can perform computations.



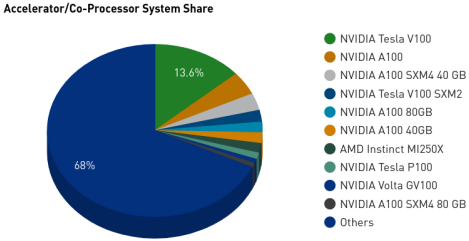


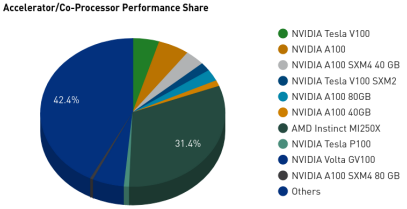
Source:Nvidia: CUDA programming

Important differences between CPU and GPU

- However, the GPU can handle many threads in parallel, which can process much data simultaneously.
- In the GPU, cores are grouped into GPU Processing Clusters (GPCs), with each GPC having its own Streaming Multiprocessors (SMs) and Texture Processor Clusters (TPCs).
- Nvidia (microarchitecture): Tesla (2006), Fermi (2010), Kepler (2012), Maxwell (2014), Pascal (2016), Volta (2017), Turing (2018), Ampere (2020), Ada Lovelace (2022), Hopper (2022), and Blackwell (2024)
- The first Nvidia CPU with a GPU (simply called superchip): Nvidia Grace Hopper
- Video Link : Mythbusters Demo GPU versus CPU

Nvidia GPU





- CUDA is a low-level programming model for Nvidia GPUs.
- HIP is a low-level programming model for AMD GPUs.

Source:https://www.top500.org

Info

HIP is similar to CUDA; therefore, knowing CUDA is an advantage for both Nvidia and AMD GPUs.

Top 5 systems

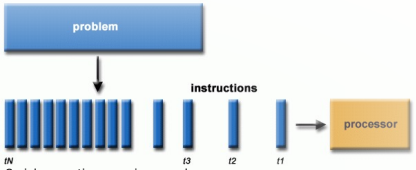
Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**El Capitan** - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, **HPE** DOE/NNSA/LLNL United States	11,039,616	1,742.00	2,746.38	29,581
2	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS, **HPE** DOE/SC/Oak Ridge National Laboratory United States	9,066,176	1,353.00	2,055.72	24,607
3	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, **Intel** DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
4	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, **Microsoft Azure** Microsoft Azure United States	2,073,600	561.20	846.84	
5	**HPC6** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, RHEL 8.9, **HPE** Eni S.p.A. Italy	3,143,520	477.90	606.97	8,461

Source: https://www.top500.org

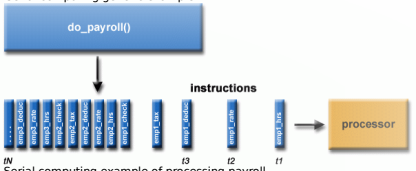
Serial programming vs parallel programming

- Serial programming
  - An entire problem can be divided into discrete series of instructions.
  - All the instructions are executed one by one.
  - Executed by a single thread or processor.
  - Only one instruction can be executed at the same time.
- Parallel programming
  - An entire problem can be divided into discrete parts in such a way that it can be solved concurrently.
  - Each part may have a set of instructions.
  - Each part's instructions are executed on a different thread/processor.
  - Since it is a parallel execution, a target problem needs to be controlled/coordinated.
- CPU, GPU, and other parallel processors can perform parallel computing.

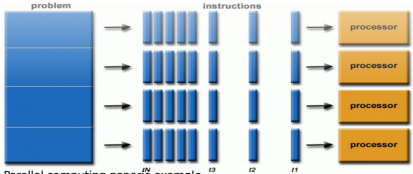
Serial programming vs parallel programming



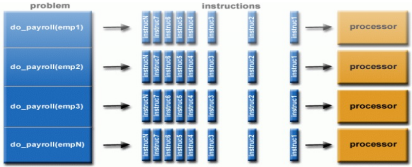
Serial computing generic example



Serial computing example of processing payroll



Parallel computing generic example



Parallel computing example of processing payroll

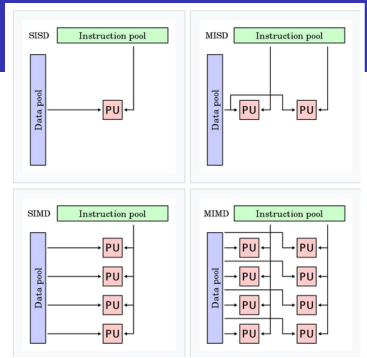
Source:HPC LLNL

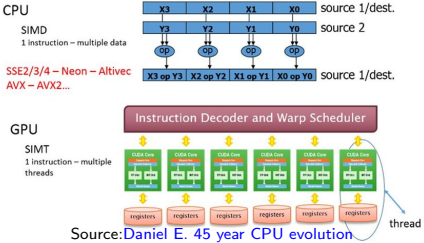
Outline for section 2

GPU architecture

- Computer architecture is characterized by four categories according to Flynn's taxonomy
  - Single instruction stream, single data stream (SISD)
  - Single instruction stream, multiple data streams (SIMD)
  - Multiple instruction streams, single data stream (MISD)
  - Multiple instruction streams, multiple data streams (MIMD)

- GPUs are based on Single Instruction Multiple Threads (SIMT)

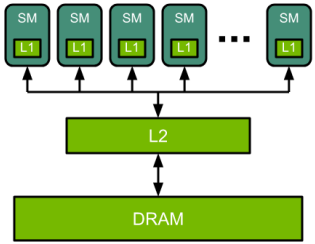




Source: Daniel E. 45 year CPU evolution

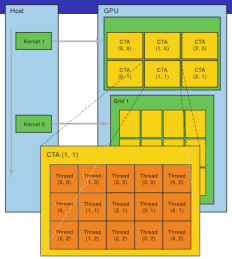
GPU architecture

- Ampere GPU has seven GPCs, 42 TPCs, and 84 SMs.
- Volta GPU has six GPCs; each GPC has seven TPCs (each including two SMs), for a total of 14 SMs.
- Each SM has L1 cache (up to 128 KB), and L2 (up to 6144 KB) cache is shared among the GPCs.
- RT (Ray Tracing) cores are dedicated to performing the ray-tracing rendering math computation.



GPU architecture

- SIMT enables programmers to achieve thread-level parallelism in streaming multiprocessors (SMs)
- The multiprocessor `occupancy` is the ratio of active warps to the maximum number of warps supported on the GPU's multiprocessor
- SMs in the GPU is based on the scalable array multi-thread, which allows grid and thread blocks of 1D, 2D, and 3D data
- Programmers can write the grid and block size to create a thread when executing the device kernel; this thread block is typically called as Cooperative Thread Array (CTA)
- A parallel execution is happening in the SMs via `warps`, and one warp contains **32 threads**



Source:Nvidia: Parallel Thread Execution

Outline for section 3

Usage of computing capabilities in different Nvidia GPU architecture

Compute capability (flag)	**Architecture support**
sm_35, and sm_37	Basic features + Kepler support + Unified memory programming + Dynamic parallelism support
sm_50, sm_52 and sm_53	+ Maxwell support
sm_60, sm_61, and sm_62	+ Pascal support
sm_70 and sm_72	+ Volta support
sm_75	+ Turing support
sm_80, sm_86 and sm_87	+ NVIDIA Ampere GPU architecture support

Further information can be found in Technical Specifications per Compute

Info

During the compilation, the compute capability is invoked as -arch=sm_70

Outline for section 4

Thread organization

- Threads are organized within Grids and Blocks. These Grids and Blocks can be in 1D, 2D or 3D. These are declared as `dim3`
- Example: 2D grid and thread block

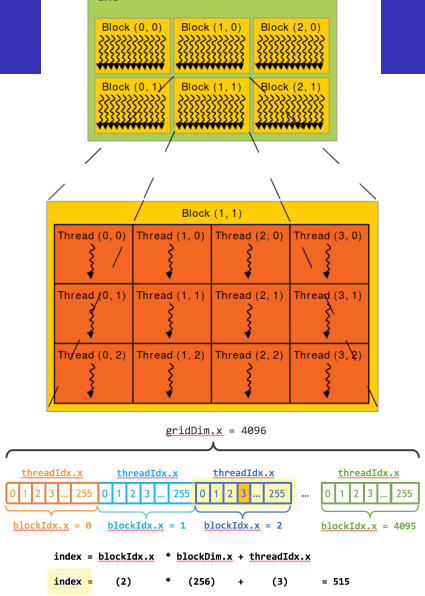
```
// two dimensional grid
dim3 Grid(3, 2, 1)
// two dimensional thread block
dim3 Block(3, 2, 1)
```

- Example: 1D grid and thread block

```
// one dimensional grid
dim3 Grid(4096, 1, 1)
// one dimensional thread block
dim3 Block(256, 1, 1)
```

- Example: calling thread block in the main program

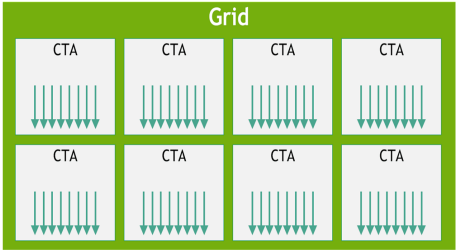
```
// calling a c/c++ function
void hello_world();
//calling cuda device function
hello_world<<<Grid, Block>>>();
```

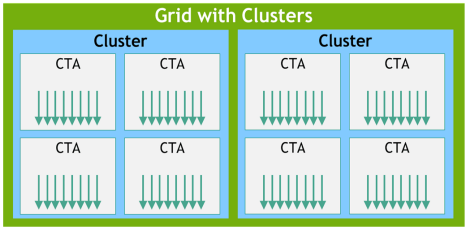


Source:Nvidia: CUDA programming

Thread Organization

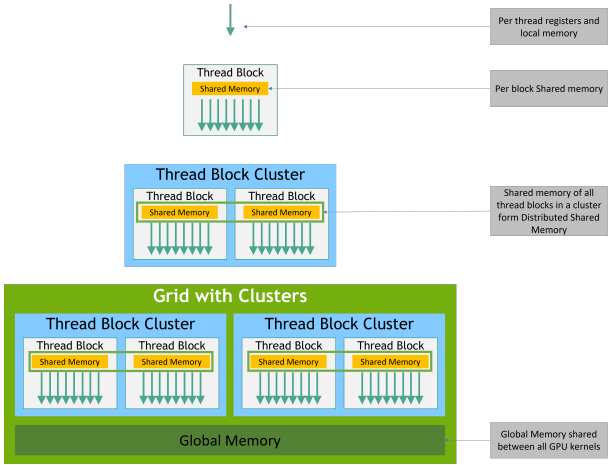
- Cooperative Thread Arrays
- Cluster of Cooperative Thread Arrays
- Grid of Clusters





Thread organization

- Dimension variables:
  - **gridDim** specifies the number of blocks in the grid.
  - **blockDim** specifies the number of threads in each block.
- Index variables:
  - **blockIdx** gives the index of the block in the grid.
  - **threadIdx** gives the index of the thread within the block.



Outline for section 5

CUDA API

- `cudaMalloc()` allocates device memory.
- `cudaMemcpy()` transfers data to or from a device.
- `cudaFree()` frees device memory that is no longer in use.
- `__syncthreads()` synchronizes threads within a block.
- `cudaDeviceSynchronize()` effectively synchronizes all threads in a grid.
- `cudaMallocManaged()` allocates unified memory.
- Memory Allocation and Lifetime

	cudaMalloc() on Host	**cudaMalloc() on Device**
cudaFree() on Host	Supported	Not Supported
cudaFree() on Device	Not Supported	Supported
Allocation limit	Free device memory	cudaLimitMallocHeapSize

Outline for section 6

Major comparison between Turing vs. Ampere

Graphics Card	GeForce RTX 2080 Founders Edition	GeForce RTX 3080 10GB Founders Edition
GPU Codename	TU104	GA102
GPU Architecture	**Nvidia Turing**	**Nvidia Ampere**
GPCs	6	6
TPCs	23	34
SMs	46	68
CUDA Cores / SM	64	128
CUDA Cores / GPU	2944	8704
Tensor Cores / SM	8 (2nd Gen)	4 (3rd Gen)
Tensor Cores / GPU	368	272 (3rd Gen)
RT cores	46 (1st Gen)	68 (2nd Gen)

Source:Nvidia Ampere

Compute capabilities for latest Nvidia GPUs

Data Center GPU	**Nvidia V100**	**Nvidia A100**	**Nvidia H100**
GPU architecture	Nvidia Volta	Nvidia Ampere	Nvidia Hopper
Compute Capability	7	8	9
Thread / Warp	32	32	32
Max Warps / SM	64	64	64
Max Threads / SM	2048	2048	2048
Max Thread Blocks (CTAs) / SM	32	32	32
Max Threads Blocks / Thread Block Clusters	NA	NA	NA
Max 32-bit Registers / SM	65536	65536	65536
Max Registers / Thread Block	65536	65536	65536
Max Registers / Thread	255	255	255
Max Thread Block Size (#of threads)	1024	1024	1024
FP32 Cores / SM	64	64	64
Ratio of SM Registers to FP32 Cores	1024	1024	1024
Shared Memory Size / SM	Configurable up to 96 KB	Configurable up to 164 KB	Configurable up to 228 KB

Outline for section 7

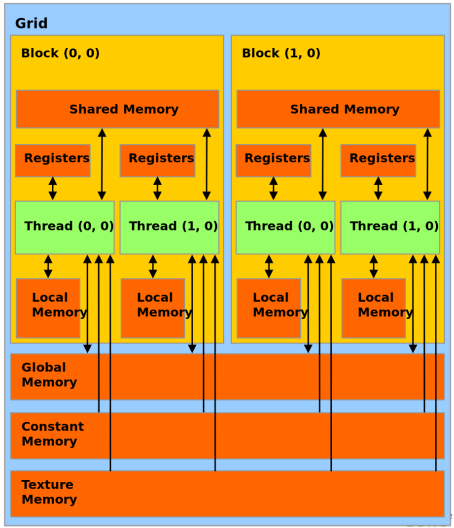
CUDA function qualifiers

Qualifier	Description
__device__	These functions are executed only from the device and callable only from device
__global__	These functions are executed from the device, and it can be callable from the host and device (only for compute capabilities 3.2 or higher)
__host__	These functions are executed from a host, and callable only from the host
__noninline__ *__forceinline__*	Compiler directives instruct the functions to be inline or not inline

An inline function (eliminates call-linkage overhead and can expose significant optimization opportunities) is explained in detail *here*.

CUDA Memory

- Registers are faster. The maximum number of 32-bit registers per thread is 255, and the maximum number of 32-bit registers per thread block is 64 K.
- Shared Memory has much higher bandwidth and much lower latency than local or global memory.
- Local Memory resides in device memory, so local memory accesses have the same high latency and low bandwidth as global memory.
- Constant memory is space that resides in device memory and is cached in the constant cache (read only).



CUDA variable memory space specifiers

Variable	Memory	Scope	Lifetime
__device__	Global	Grid (entire grid of thread blocks)	Application
__constant__	Constant	Grid (entire grid of thread blocks)	Application
__shared__	Shared	Block (within a thread block)	Block

- `__device__` int DeviceVar;
  Resides in global memory space
- `__constant__` int ConstantVar;
  Resides in constant memory space and is read-only (cached)
- `__shared__` int SharedVar;
  Resides in the shared memory space of a thread block

Outline for section 8

Hello world

- Run a part or the entire application on the GPU
- Call `cuda_function` on the device
- It should be called using the function qualifier `__global__`
- Calling the device function in the main program:
  - C/C++ example, `c_function()`
  - CUDA example, `cuda_function<<<1,1>>>()` (just using 1 thread)
- `<<< >>>`, specify the thread blocks within the brackets
- Make sure to synchronize the threads
  - `__syncthreads()` synchronizes all the threads within a thread block
  - `CudaDeviceSynchronize()` synchronizes a kernel call on the host
- Most CUDA APIs are synchronized calls by default (but sometimes it's good to call explicit synchronized calls to avoid errors in computation)

Example: Hello world

C version

```c
#include <studio.h>
void c_function()
{
  printf{"Hello World!\n"};
}

int main()
{
  c_function();
  return 0;
}
```

CUDA version

```c
#include <studio.h>
__global__ void cuda_function()
{
  printf{"Hello World from GPU!\n"};
  __syncthreads();
}

int main()
{
  cuda_function<<<1,1>>>();
  cudaDeviceSynchronize();
  return 0;
}
```

Outline for section 9

Example: Vector addition

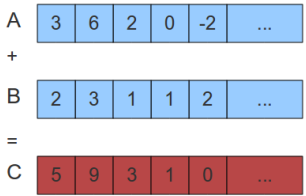
- Memory allocation on both CPU and GPU

```
// Initialize the memory on the host
float *a, *b, *out;

// Allocate host memory
a   = (float*)malloc(sizeof(float) * N);
b   = (float*)malloc(sizeof(float) * N);
out = (float*)malloc(sizeof(float) * N);

// Initialize the memory on the device
float *d_a, *d_b, *d_out;

// Allocate device memory
cudaMalloc((void**)&d_a, sizeof(float) * N);
cudaMalloc((void**)&d_b, sizeof(float) * N);
cudaMalloc((void**)&d_out, sizeof(float) * N);
```



Example: Vector addition

- Fill values for host vectors a and b

```
// Initialize host arrays
for(int i = 0; i < N; i++)
   {
    a[i] = 1.0f;
    b[i] = 2.0f;
   }
```

- Transfer initialized value from CPU to GPU

```
// Transfer data from host to device memory
cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, sizeof(float) * N, cudaMemcpyHostToDevice);
```

- Creating a 2D thread block

```
// Thread organization
dim3 dimGrid(1, 1, 1);
dim3 dimBlock(32, 32, 1);
```

- Calling the kernel function

```
// execute the CUDA kernel function
vector_add<<<dimGrid, dimBlock>>>(d_a, d_b, d_out, N);
```

Example: Vector addition

- Copy back computed value from GPU to CPU

```
// Transfer data back to host memory
cudaMemcpy(out, d_out, sizeof(float) * N, cudaMemcpyDeviceToHost);
```

- Vector addition function call

```
  // GPU function that adds two vectors
__global__ void vector_add(float *a, float *b,
           float *out, int n)
{
  int i = blockIdx.x * blockDim.x * blockDim.y +
    threadIdx.y * blockDim.x + threadIdx.x;
  // Allow the   threads only within the size of N
  if(i < n)
    {
      out[i] = a[i] + b[i];
    }
  // Synchronise all the threads
  __syncthreads();
}
```

Example: Vector addition

- Release the host and device memory

```
// Deallocate device memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_out);

// Deallocate host memory
free(a);
free(b);
free(out);
```

Source: Vector-Addition.cu

Outline for section 10

Example: Matrix multiplication

Matrix multiplication function in C/C++

```
float * matrix_mul(float *h_a, float *h_b,
                   float *h_c, int width)
{
  for(int row = 0; row < width ; ++row)
    {
      for(int col = 0; col < width ; ++col)
        {
          float single_entry = 0;
          for(int i = 0; i < width ; ++i)
            {
              single_entry += h_a[row*width+i]
              * h_b[i*width+col];
            }
          h_c[row*width+col] = single_entry;
        }
    }
  return h_c;
}
```

Source:Matrix-Multiplication.cu

Matrix multiplication function in CUDA

```
__global__ void matrix_mul(float* d_a, float* d_b,
                           float* d_c, int width)
{
  int row = blockIdx.x * blockDim.x + threadIdx.x;
  int col = blockIdx.y * blockDim.y + threadIdx.y;

  if ((row < width) && (col < width))
    {
      float single_entry = 0;
      // each thread computes one
      // element of the block sub-matrix
      for (int i = 0; i < width; ++i)
        {
          single_entry += d_a[row*width+i] *
                          d_b[i*width+col];
        }
      d_c[row*width+col] = single_entry;
    }
}
```

Example: Matrix multiplication

- Allocating the CPU and GPU memory for A, B, and C matrix

```
// Initialize the memory on the host
float *a, *b, *c;

// Initialize the memory on the device
float *d_a, *d_b, *d_c;

// Allocate host memory
a = (float*)malloc(sizeof(float) * (N*N));
b = (float*)malloc(sizeof(float) * (N*N));
c = (float*)malloc(sizeof(float) * (N*N));

// Allocate device memory
cudaMalloc((void**)&d_a, sizeof(float) * (N*N));
cudaMalloc((void**)&d_b, sizeof(float) * (N*N));
cudaMalloc((void**)&d_c, sizeof(float) * (N*N));
```

- Transfer initialized A and B matrix from CPU to GPU

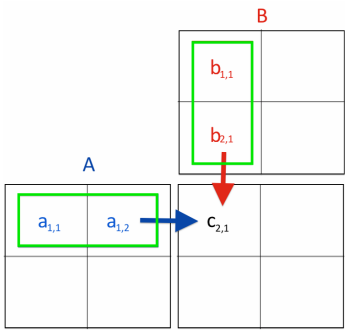
```
// Transfer data from a host to device memory
cudaMemcpy(d_a, a, sizeof(float) * (N*N), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, sizeof(float) * (N*N), cudaMemcpyHostToDevice);
```

- Calling the kernel function

```
// Device function call
matrix_mul<<<dimGrid,dimBlock>>>(d_a, d_b, d_c, N);
```

- 2D thread block for indexing x and y

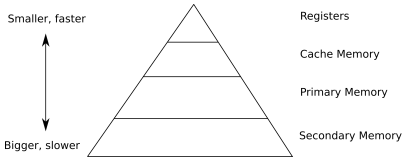
```
// Thread organization
int blockSize = 32;
dim3 dimBlock(blockSize,blockSize,1);
dim3 dimGrid(ceil(N/float(blockSize)),
             ceil(N/float(blockSize)),1);
```

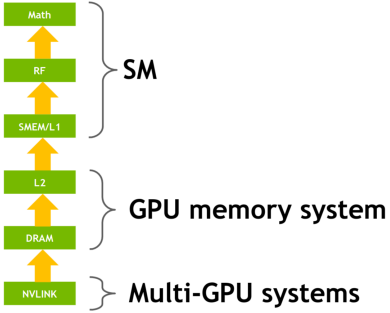


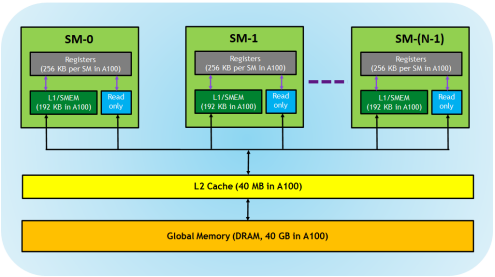
Shared Memory

- Shared memory is declared as `__shared__` int SharedVar;
- Keeping the data closer to the CUDA cores makes computation faster.
- On Nvidia GPUs, L1 cache and shared memory can be reconfigured by using the CUDA API `cudaFuncSetCacheConfig();`
  - cudaFuncCachePreferNone: no preference for shared memory or L1 (default)
  - cudaFuncCachePreferShared: prefer larger shared memory and smaller L1 cache
  - cudaFuncCachePreferL1: prefer larger L1 cache and smaller shared memory
  - cudaFuncCachePreferEqual: prefer equal size L1 cache and shared memory
- Advanced topic: `tuning for L2 cache`, quite useful when multiple kernels need to use the same data. In that case, we can keep the frequently used data block (by multiple kernels) in the L2 cache instead of accessing global memory.

Shared Memory







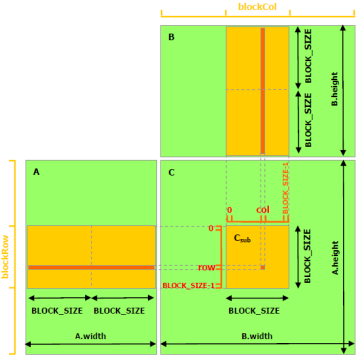
Shared Memory: Matrix Multiplication

- Matrix function call

```
// Device call (matrix multiplication)
__global__ void matrix_mul(const float *d_a,
const float *d_b, float *d_c, int width)
{// Shared memory allocation for the block matrix
  __shared__ int a_block[BLOCK_SIZE][BLOCK_SIZE];
  __shared__ int b_block[BLOCK_SIZE][BLOCK_SIZE];
  // Indexing for the block matrix
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  // Indexing global matrix to block matrix
  int row = threadIdx.x+blockDim.x*blockIdx.x;
  int col = threadIdx.y+blockDim.y*blockIdx.y;
  // Allow threads only for size of rows and columns (we assume square matrix)
  if ((row < width) && (col< width))
    {// Save temporary value for the particular index
      float temp = 0;
      for(int i = 0; i < width / BLOCK_SIZE; ++i)
       {
       // Allign the global matrix to block matrix
       a_block[ty][tx] = d_a[row*width+(i*BLOCK_SIZE+tx)];
       b_block[ty][tx] = d_b[(i*BLOCK_SIZE+ty)* width+col];
       // Make sure all the threads are synchronized
       __syncthreads();
       // Multiply the block matrix
       for(int j = 0; j < BLOCK_SIZE; ++j)
       temp += a_block[ty][j] * b_block[j][tx];
       __syncthreads();
       }
      // Save block matrix entry to global matrix
```

- 2D thread block for indexing x and y

```
// Thread organization
  dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE, 1);
  dim3 dimGrid(ceil(N/BLOCK_SIZE),
  ceil(N/BLOCK_SIZE), 1);
```



Outline for section 11

Unified Memory

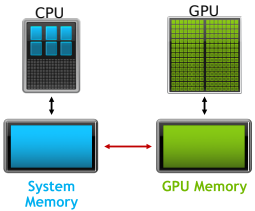
Without unified memory

- Allocate the host memory
- Allocate the device memory
- Initialize the host value
- Transfer the host value to the device memory location
- Do the computation using the CUDA kernel
- Transfer the data from the device to the host
- Free device memory
- Free host memory

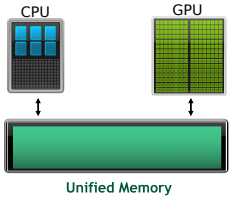
With unified memory

- ~~Allocate the host memory~~
- Allocate the device memory
- Initialize the host value
- ~~Transfer the host value to the device memory location~~
- Do the computation using the CUDA kernel
- ~~Transfer the data from the device to the host~~
- Free device memory
- ~~Free host memory~~

Without unified memory concept



With unified memory concept



Source:Vector-Addition-Unified.cu

Example: Unified Memory - Vector addition

Use `cudaMallocManaged()`

```
/*
// Initialize the memory on the host
float *a, *b, *out;
// Allocate host memory
a = (float*)malloc(sizeof(float) * N);
b = (float*)malloc(sizeof(float) * N);
out = (float*)malloc(sizeof(float) * N);
*/

// Initialize the memory on the device
float *d_a, *d_b, *d_out;

// Allocate device memory
cudaMallocManaged(&d_a, sizeof(float) * N);
cudaMallocManaged(&d_b, sizeof(float) * N);
cudaMallocManaged(&d_out, sizeof(float) * N);
```

Do not forget to call `cudaDeviceSynchronize()` after a kernel call

Source: Vector-Addition-Unified.cu

Performance and Profiling

Using the CUDA API, we can measure the time taken to execute the CUDA kernel functions. The example below shows how to measure the time taken for a CUDA kernel.

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);

// Device function call
matrix_mul<<<Grid_dim, Block_dim>>>(d_a, d_b, d_c, N);

//use CUDA API to stop the measuring time
cudaEventRecord(stop);
cudaEventSynchronize(stop);
float time;
cudaEventElapsedTime(&time, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);

cout << " time taken for the GPU kernel" << time << endl;
```

Performance and Profiling

`Nvidia system-wide performance analysis` will help to analyze the code where time is spent, for example, in computation and communication.
`Nvidia` provides profiling tools and can measure the traces and events of the CUDA application.
Nvidia HPC SDK provides the following profiling tools for the CUDA application

- `Nsight Compute:` Profile the kernel calls; both `visual profile-GUI` and `Command Line Interface` can be used to check the profiling information of the kernel calls.
- `Nsight Graphics:` This is quite useful for analyzing the profiling results through the GUI.
- `Nsight Systems:` Provides system-wide profiling; for example, when the application involves mixed programming with CPU and GPU (that is, MPI, OpenMP, and CUDA).

Outline for section 12

More and advanced topics

1. CUDA streams
2. OpenACC
3. OpenMP offloading
4. HIP for AMD GPUs

OpenACC
Directives for Accelerators

OpenMP

AMD

Outline for section 13

Further information and resources

CUDA books archive

MOOC online course (PRACE) - GPU Programming for Scientific Computing and Beyond: Covering CUDA and OpenACC (from basics to advanced) - Please contact me for more information.

Introduction to OpenACC, OpenMP Offloading, and HIP programming models (will be available since November 2024) - Please contact me for more information.

OpenMP Programming model (May 2025 - NCC Luxembourg) - Please contact me for more information.

OpenACC Programming model (June 2025 - NCC Luxembourg) - Please contact me for more information.