CUDA: understand to be a genuine user

takigawa

May 22, 2025

The university of Tokyo, EEIC, Taura Lab

Contents

1. What is CUDA?: Introduction
2. A CUDA program for beginners
3. How CUDA works
4. Optimize CUDA program
5. Practice Problems

What is CUDA?

Contents

1. What is CUDA?: Introduction
2. A CUDA program for beginners
3. How CUDA works
4. Optimize CUDA program
5. Practice Problems

CUDA is the abstraction of GPU(s) for programmers

GPU (Graphical Processing Unit) is a device separated from CPU (Central PU).

- Code that runs on GPU must be designated as a kernel
- Data must be copied between CPU and GPU(s)
- A GPU is often called a *"device"*
- A CPU is often called a *"host"*

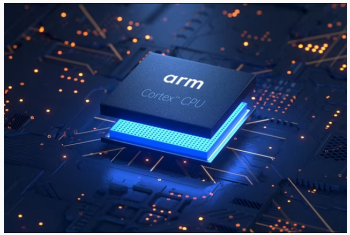


Figure 1: Host

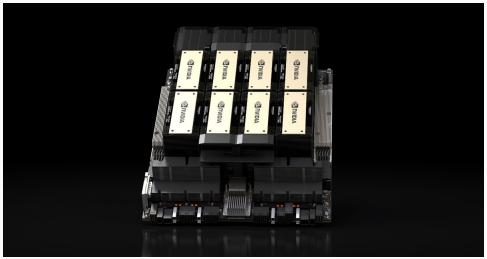


Figure 2: Device

CUDA is the abstraction of GPU(s) for programmers

CUDA is a platform for parallel computing on NVIDIA GPU.

- language extension: C/C++, Fortran
- tools: compiler(nvcc), debugger(cuda-gdb), profiler(Nsight Systems)
- APIs: Driver API, Runtime API
- Libraries: cuBLAS, cuFFT, cuDNN, etc.

Their common goal is to provide programmers with a "good" abstraction of GPU(s). How "good"?: usable, simple, highly affine to hardware(=easy to bring out the performance)

To compile/run CUDA programs: NVCC

```
nvcc -arch=sm_90 programs.cu%
```

- compile with `nvcc` command
- the natural extension of CUDA program is `.cu`
- `-arch` flag designates GPU Architecture
  - `compute_XX`: Virtual architecture
  - `sm_XX`: Physical architecture (SM generation)

Interlude: How to know the proper architechture

Use `cudaGetDeviceXXXXX` APIs and get device query.

`device_query.cu`

```
#include <cuda_runtime.h>
int deviceCount;
cudaError_t error = cudaGetDeviceCount(&deviceCount);
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, i);

printf("Device %d: %s\n", i, deviceProp.name);
printf("  Compute capability: %d.%d\n", deviceProp.major, deviceProp.minor);
printf("  Total global memory: %.2f GB\n", deviceProp.totalGlobalMem / (1024.0
     * 1024.0 * 1024.0));
```

or

```
  nvidia-smi --query-gpu=compute_cap
```

The architecture for GH200 is `sm_90`.

A CUDA program for beginners

Contents

1. What is CUDA?: Introduction
2. A CUDA program for beginners
   2.1 Kernels; writing and launching
   2.2 Host-Device Data Communication (+ Synchronization)
3. How CUDA works
4. Optimize CUDA program
5. Practice Problems

Setting environment: Using GH200(s) on miyabi

1. `https://miyabi-www.jcahpc.jp/login` にアクセスし、パスワード初期化を選択する
2. 指示にしたがい、 Miyabi 利用支援ポータルにアクセスする
3. ドキュメント閲覧/Miyabi システム利用手引書 をダウンロードする (Strongly recommended)
4. 手引書の P.9 システム初回ログイン時の設定 の手順を完了する
5. 手引書の P.22 SSH ログイン/初回ログイン の手順を完了する (必ず緊急用スクラッチコードを控えること)
6. (必要に応じて) エディタから 2 回目ログインを行う

2 要素認証が必須である。

【注意】 ログインノード `/home/cXXXXX`ではなく、計算ノード`/work/gc64/cXXXXX`で作業する

Sample program #1: hello_world.cu

See https://github.com/gunnersgoestocl/cuda-introduction/tree/main/tutorial-legacy for more information.

```
1  #include <stdio.h>
2  #include <cuda_runtime.h>              // for Runtime APIs
3
4  __global__ void hello(){               // kernel function
5      printf("Hello CUDA World !!\n");
6  }
7
8  int main() {
9      hello<<< 2, 4 >>>();               // launch kernel
10     cudaDeviceSynchronize();           // wait until kernel completes
11     return 0;
12 }
13
```

3 files are required for execution on miyabi

- `.cu` file: CUDA user program
- `makefile`: compile and clean
- shell script: to submit batch job, see official docs for more info

```
NVCC := nvcc
NVCCFLAGS := -arch=sm_90 -O3
# .cu ファイルから実行ファイルを生成
CUDA_EXECUTABLES := $(patsubst %.cu,%,$(wildcard
      *.cu))
# デフォルトのターゲット
all: $(C_EXECUTABLES) $(CUDA_EXECUTABLES)
# .cu ファイルから実行ファイルを生成
%: %.cu
^^I$(NVCC) $(NVCCFLAGS) $< -o $@
# clean ターゲットの定義
.PHONY: clean
clean:
^^Irm -f $(CUDA_EXECUTABLES)
```

```
#!/bin/bash
#PBS -q debug-g
#PBS -l select=1
#PBS -W group_list=gc64
#PBS -j oe

module purge
module load cuda

cd ${PBS_O_WORKDIR}
./a.out 256
```

Contents of .cu file: kernel function

- "kernel" (sometimes "GPU kernel"): A function that runs on GPU
  - SYNTAX: An ordinary C/C++ function that returns nothing (`void`)
  - SYNTAX: Add `__global__` keyword beforehand

```
1 __global__ void f(...args...) { ...body... }
2
```

Listing 1: kernel template

Contents of .cu file: launching kernel by a host

A host (CPU) launches a kernel to devices.

- Programmers must specify the number of threads by `<<nb, bs>>`
  - `nb`: Number of Blocks (per grid) (sometimes `gridDim`)
  - `bs`: Block Size (sometimes `blockDim`)
- `nb * bs` is the number of CUDA threads created

```
1    // ... code run on host ...
2
3    f<<gridDim, blockDim>>(...args...);
```

- `nb`, `bs` can be $1, 2, 3$-Dimensional using type `dim3`

```
1  dim3 block(x_threads_block, y_threads_block);     // x, y(, z)
2  dim3 grid(x_blocks_grid, z_blocks_grid);          // x, y(, z)
3
4  f<<grid, block>>(...args...);
```

Interlude: register values programs can explicitly use

- Threads which executes a single instruction can be executed in parallel.
- So, programmers are expected to make the kernel visible to all threads as the same instruction.
- For this perspective, a unique ID of each thread (= the loop index) is the key.

In CUDA, each kernel can access its own thread ID through built-in variables on the Special Registers.

- `blockDim.{x,y} = bs` (the block size)
- `gridDim.{x,y} = nb` (the number of blocks)
- `threadIdx.{x,y} =` the thread ID within a block ($\in [0, bs)$)
- `blockIdx.{x,y} =` the thread's block ID ($\in [0, nb)$)

-¿ `blockDim.x * blockIdx.x + threadIdx.x` could be the loop index

Sample program #2: hello_gpu.cu

```
1  #include <stdio.h>
2  #include <cuda_runtime.h>
3
4  __device__ void gpuAdd(int *number){  *number += 1; }
5  __global__ void callGpu(int *number){  gpuAdd(number); }
6
7  int main(){
8    int device_id = 0;  cudaSetDevice(device_id); //device set up
9    int *a = (int*)malloc(sizeof(int));  *a = 0;  //allocate memory on host(cpu)
10   int *a_dev = 0;  cudaMalloc((void**)&a_dev, sizeof(int)); // allocate memory on gpu
11   cudaMemcpy(a_dev, a, sizeof(int), cudaMemcpyHostToDevice);  // memcpy host -> device
12   // execute
13   callGpu<<<1, 1>>>(a_dev);
14   cudaDeviceSynchronize(); // Wait until GPU processing finishs.
15
16   cudaMemcpy(a, a_dev, sizeof(int), cudaMemcpyDeviceToHost);  // memcpy device -> host
17   cudaFree(a_dev);  // free
18
19   printf("ans: %d \n", *g); return 0; // display the answer
20 }
```

Data communication between H & Ds: overview

- Host memory and device memory are (basically) *separete*
- The device(D) cannot access data on the host(H) and vice versa by hardware
  - Access to another memory (including that of another device) causes `Segfault`
- Software need to explicitly specify when and what to communicate between H & D

Data communication between H & Ds: template to send

1. allocate data of the same size both on host and device

```
size_t sz = sizeof(int)*len;
int *a = (int*)malloc(sz);
int *a_dev = 0;  cudaMalloc((void**)&a_dev, sz);
// (void**)&a_dev is the address of the pointer variable (a_dev)
// Head address of GPU global memory is written to a_dev
```

Data communication between H & Ds: template to send

1. allocate data of the same size both on host and device

```
size_t sz = sizeof(int)*len;
int *a = (int*)malloc(sz);
int *a_dev = 0;  cudaMalloc((void**)&a_dev, sz);
// (void**)&a_dev is the address of the pointer variable (a_dev)
// Head address of GPU global memory is written to a_dev
```

2. the host works on the kinda initialization of the host data

```
for ( ... ) { a[i] = ... } // on host, initialize input data of kernel
```

Data communication between H & Ds: template to send

1. allocate data of the same size both on host and device

```
size_t sz = sizeof(int)*len;
int *a = (int*)malloc(sz);
int *a_dev = 0;  cudaMalloc((void**)&a_dev, sz);
// (void**)&a_dev is the address of the pointer variable (a_dev)
// Head address of GPU global memory is written to a_dev
```

2. the host works on the kinda initialization of the host data

```
for ( ... ) { a[i] = ... } // on host, initialize input data of kernel
```

3. copy the data to the device

```
cudaMemcpy(a_dev, a, sz, cudaMemcpyHostToDevice); // target address,
    source address, size, keyword
```

Data communication between H & Ds: template to send

1. allocate data of the same size both on host and device

```
size_t sz = sizeof(int)*len;
int *a = (int*)malloc(sz);
int *a_dev = 0;  cudaMalloc((void**)&a_dev, sz);
// (void**)&a_dev is the address of the pointer variable (a_dev)
// Head address of GPU global memory is written to a_dev
```

2. the host works on the kinda initialization of the host data

```
for ( ... ) { a[i] = ... } // on host, initialize input data of kernel
```

3. copy the data to the device

```
cudaMemcpy(a_dev, a, sz, cudaMemcpyHostToDevice); // target address,
    source address, size, keyword
```

4. pass the device pointer to the kernel

```
f<<nb, bs>>(a_dev, ...);
```

Data communication between H & Ds: template to retrieve

1. allocate data of the same size both on host and device

```
size_t sz = sizeof(int)*len;
int *r = (int*)malloc(sz);                          // host memory
int *r_dev = 0;  cudaMalloc((void**)&r_dev, sz);   // device memory
```

Data communication between H & Ds: template to retrieve

1. allocate data of the same size both on host and device

```
size_t sz = sizeof(int)*len;
int *r = (int*)malloc(sz);                          // host memory
int *r_dev = 0;  cudaMalloc((void**)&r_dev, sz);   // device memory
```

2. pass the device pointer of receptacle to the kernel

```
f<<nb, bs>>(...,r_dev, ...);  // args must include pointer(s) of input
    and output
```

Data communication between H & Ds: template to retrieve

1. allocate data of the same size both on host and device

```
size_t sz = sizeof(int)*len;
int *r = (int*)malloc(sz);                          // host memory
int *r_dev = 0;  cudaMalloc((void**)&r_dev, sz);   // device memory
```

2. pass the device pointer of receptacle to the kernel

```
f<<nb, bs>>(...,r_dev, ...);  // args must include pointer(s) of input
    and output
```

3. copy the data to the device

```
cudaMemcpy(r, r_dev, sz, cudaMemcpyDeviceToHost); // target address,
    source address, size, keyword
```

Host-Device synchronization

- A kernel call and the host overlap.
- Multiple kernel calls are serialized on the GPU side, by default (Host basically cannot control)
  - Grid is an abstraction of a one time launch of the kernel.
  - Grid is managed using a data struct FIFO queue called Stream
  - If you want to execute multiple kernel calls concurrently, you must use multiple Streams or Devices.
- `cudaDeviceSynchronize()` is an API to wait for the kernel to finish

```
1 h0();
2 g0<<...,...>>();
3 h1();
4 g1<<...,...>>();
5 cudaDeviceSynchronize();
6 h2();
```

- `g0` might overlap with `h1`
- `g0` and `g1` do not overlap because they are assigned to the same stream 0
- `h2`s does not overlap with anything because of `cudaDeviceSynchronize()`

(+) "Programmer-free" data communication between H & Ds

- Recent NVIDIA GPUs support Unified Memory that **eliminate** the need for

Sample Code: `vecadd`

(+) "Programmer-free" data communication between H & Ds

- Recent NVIDIA GPUs support Unified Memory that **eliminate** the need for
  - explicit data movement between host and device memory

Sample Code: `vecadd`

(+) "Programmer-free" data communication between H & Ds

- Recent NVIDIA GPUs support Unified Memory that **eliminate** the need for
  - explicit data movement between host and device memory
  - dual pointer management

Sample Code: `vecadd`

(+) "Programmer-free" data communication between H & Ds

- Recent NVIDIA GPUs support Unified Memory that **eliminate** the need for
  - explicit data movement between host and device memory
  - dual pointer management
- cudaMallocManaged is an API that hides the copy of data between the host and devices

Sample Code: `vecadd`

(+) "Programmer-free" data communication between H & Ds

- Recent NVIDIA GPUs support Unified Memory that **eliminate** the need for
  - explicit data movement between host and device memory
  - dual pointer management
- cudaMallocManaged is an API that hides the copy of data between the host and devices
  - The CPU and GPU page tables are linked at the unified virtual address, and a hardware page fault fires the moment a GPU/CPU accesses a page.

Sample Code: `vecadd`

(+) "Programmer-free" data communication between H & Ds

- Recent NVIDIA GPUs support Unified Memory that **eliminate** the need for
  - explicit data movement between host and device memory
  - dual pointer management
- cudaMallocManaged is an API that hides the copy of data between the host and devices
  - The CPU and GPU page tables are linked at the unified virtual address, and a hardware page fault fires the moment a GPU/CPU accesses a page.
  - The moment a GPU/CPU accesses a page, a hardware page fault fires and the page is moved on-demand.

Sample Code: `vecadd`

(+) "Programmer-free" data communication between H & Ds

- Recent NVIDIA GPUs support Unified Memory that **eliminate** the need for
  - explicit data movement between host and device memory
  - dual pointer management
- cudaMallocManaged is an API that hides the copy of data between the host and devices
  - The CPU and GPU page tables are linked at the unified virtual address, and a hardware page fault fires the moment a GPU/CPU accesses a page.
  - The moment a GPU/CPU accesses a page, a hardware page fault fires and the page is moved on-demand.
  - Coherency is ensured at the kernel synchronization point.

Sample Code: `vecadd`

Conventional vs Unified Memory

Purpose	Conventional (`cudaMalloc`+`cudaMemcpy`)	Unified Memory (`cudaMallocManaged`)
Memory Management	allocates **separate buffers** for host and device, and makes copy explicit	**single pointer** can be referenced from either CPU/GPU
copy	All buffer transfers each time `cudaMemcpy()` is called	**automatic transfer per page** (on demand)
address space	different values for CPU and GPU	**Unified Virtual Address (UVA)**-share same value
oversubscribe	impossible	GPU Can allocate more than the memory capacity and swap unused pages to the host
Optimization API	None	Manual tuning of placement with `cudaMemPrefetchAsync`, `cudaMemAdvise`

HOW CUDA works

Contents

1. What is CUDA?: Introduction
2. A CUDA program for beginners
3. How CUDA works
   - 3.1 Architecture of NVIDIA GPU
   - 3.2 Grid, block, thread; abstractions by CUDA
   - 3.3 Warp; Parallel Thread eXecution
   - 3.4 Stream: beyond Grid
   - 3.5 Memory Hierarchy in CUDA
   - 3.6 Resolving race condition on CUDA
4. Optimize CUDA program
5. Practice Problems

Architecture of NVIDIA GPU: GPU unit

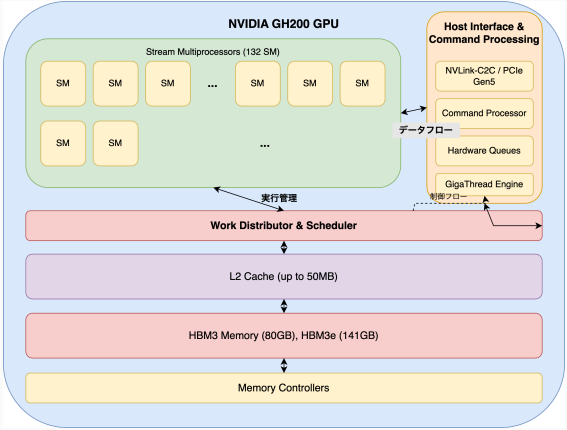


Figure 3: Device GPU

- GBM3 Memory

  known as global memory, which has large capacity but slow access speed
- Stream Multiprocessor (SM)

  In charge of multiple blocks, performs the operations that make up the kernel in parallel.
- Host Interface & Command Processing

  Interface to communicate with the host CPU, and a command processing unit that manages the execution of commands.

Architecture of NVIDIA GPU: Stream Multiprocessors

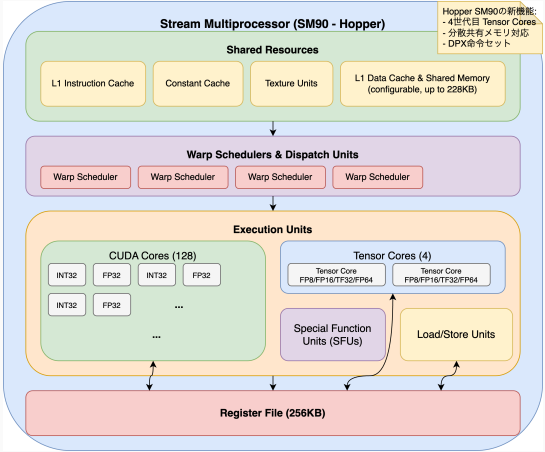


Figure 4: Device GPU

- shared memory

  has small capacity but relatively fast access speed
- Warp scheduler

  Manage parallel execution on CUDA cores in units of Warps that comprise blocks allocated to the SM
- CUDA core: The core that performs the actual computation
- Tensor core: The core specialized for matrix operations

3 easy pieces about hardware - software

Programmers know 3 things about CUDA: Grid, Block, Thread

- (Review) Grid corresponds to a one time launch of the kernel.
  - A Grid is assigned to a single GPU unit (i.e., a single device)
  - (Review) Grid is a collection of Blocks
- A Block is the unit of dispatching to an SM
  - A Block is assigned to a single SM
    i.e., once a block starts running, it stays on the SM (= occupies registers and shared memory) all the way until it finishes
  - (Review) A Block is a collection of Threads
- A Thread is the smallest unit of execution
- A Thread is assigned to a single CUDA core

Motivation: You may have questions like ...

1. Is it allowed for a `block` to have more `thread`s than the number of **processors(CUDA cores)** in the allocated **SM**, and if so, how is this handled?
2. Is it allowed to have more `block`s than the number of **SM**s that make up the **GPU unit** corresponding to the `grid`, and if so, how is this handled?
3. How is the execution across multiple `grid`s handled?

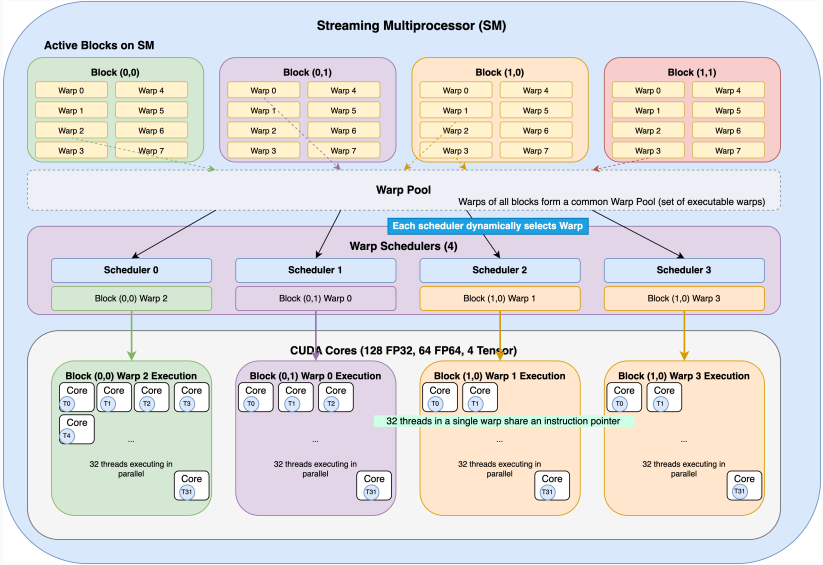
Hints:

- `thread`s belonging to the same `block` are executed by the same **SM**, this does NOT mean they are executed in parallel.
- A **SM** is assigned to a `block`, but this does NOT mean that an **SM** can only be in charge of one `block`.

Warp: The way to realize "Parallel" Thread eXecution

- The unit of instruction execution in the SM is a **Warp**
- The number of threads that make up a warp has always been **32**.
- Each thread in a warp shares an instruction pointer (i.e, executes the same instruction at the same time).
- The warp scheduler selects a warp from the ready queue (Warp pool) and executes the instruction.

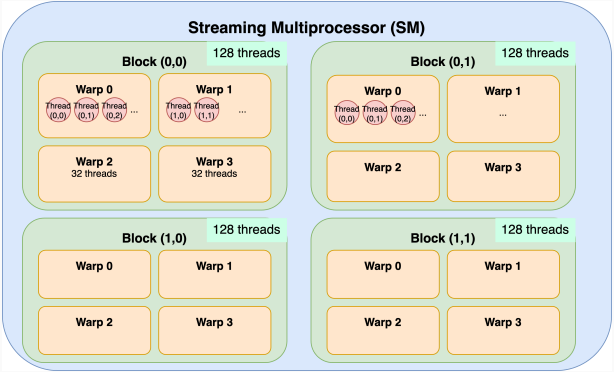
Warp management: overview



Hierarchy within an SM

Parallelism within an Stream Multiprocessor consists of three levels.

thread $\subset$ warp $\subset$ block $\subset$ SM



- (recap) A group of 32 CUDA threads makes a warp
- A group of $bs/32$ warps makes a block
- There are multiple blocks active on a single SM

Limitation of Hardware: performance degradation

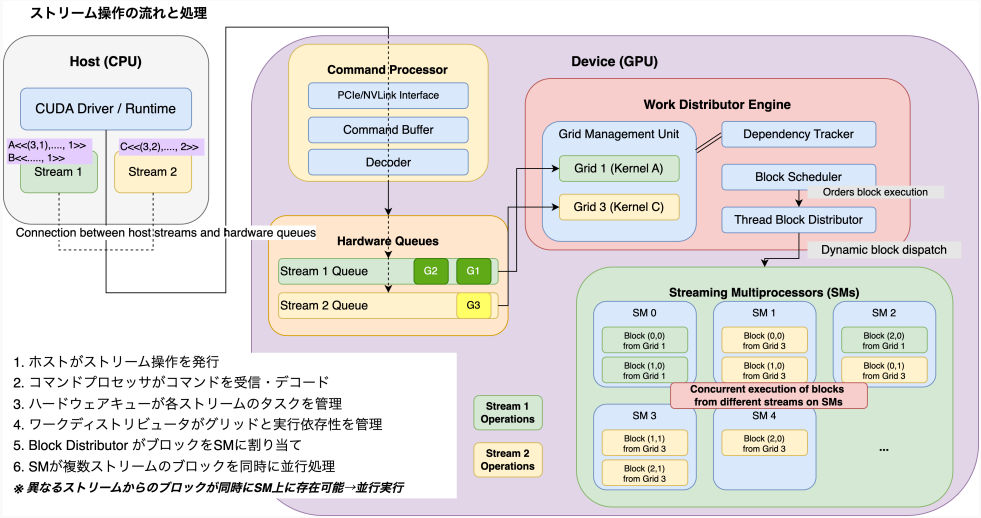
Stream

(review) As long as multiple kernels are submitted to the same stream(i.e., the default stream), they are always executed in series on the GPU side.

∵ If program doesn't specify Stream when launching a kernel, the grid is automatically assigned to legacy stream 0.

- Stream is
  a **FIFO queue** that binds the sequence of operations passed by the host to the GPU.
  > *The host runtime writes the operation to the command buffer and queues up entries with the same stream ID ; the GPU grabs the queue on doorbell notification and distributes it to the hardware engine (Device), keeping each stream in order.*
  - Grid (= a task) is assigned to a Stream.
  - If program assign grids to multiple Streams (= launch kernels), GPU firmware pop an item from the available queue with the highest priority
- Scheduling policy between Streams is a complete **blackbox**.

Flow of Stream Operation



Interlude: template to use multiple Streams

```
cudaStream_t sCompute, sCopy;
cudaStreamCreate(&sCompute);
cudaStreamCreate(&sCopy);

// 1) 非同期コピー (→) をHD stream sCopy
cudaMemcpyAsync(d_in,  h_in,  bytes, cudaMemcpyHostToDevice, sCopy);

// 2) カーネルを stream sCompute
myKernel<<<grid, block, 0, sCompute>>>(d_in, d_out);  // 0: sharedMem ID,

// 3) 非同期コピー (→) をDH stream sCopy
cudaMemcpyAsync(h_out, d_out, bytes, cudaMemcpyDeviceToHost, sCopy);

// 4) 任意の同期点
cudaEvent_t done;  cudaEventCreate(&done);
cudaEventRecord(done, sCopy);            // sCopy 完了後に立つ
cudaStreamWaitEvent(sCompute, done, 0); // sCompute は done まで待つ
```

(+) Using multiple Devices in a single Node : thread

† You CANNOT try this on Miyabi, because each node has only 1 device.

サンプルコード

However, I don't have the environment where the program can execute.

(+) Using multiple nodes: MPI

Simple Idea:

- Create a dedicated MPI process for each GPU, and process all communications with MPI
- Root process: Initialization, Result presentation
  - Activate communication, get process ID (rank), set gpu device
    - `MPI_Init, MPI_Comm_rank, MPI_Comm_size`
  - Distribute data from Root to the processes
  - Gather result from the processes to Root

Sample Program: `matmul_multinode.cu` (github link) (just `make`)

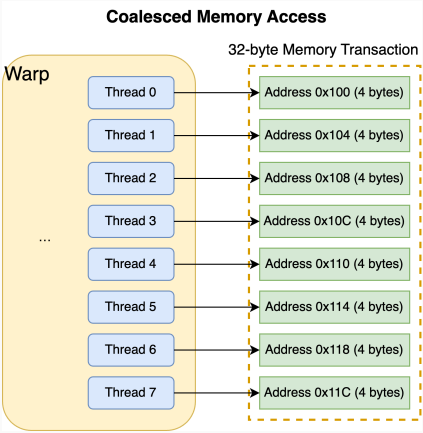
Job Script: Sample Script (github link) (just `qsub ./run.sh`)

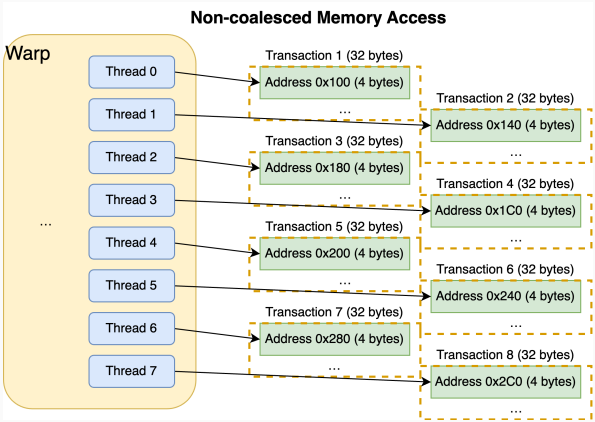
Memory hierarchy of NVIDIA GPU

To reduce bottlenecks : coalescing memory access

- DRAM (global memory) transaction size is 32bytes (i.e., Load/Store Unit handles read/write in 32-byte units with `write-enable` bits)
  - Reccomendation: refer to appendix (Hardware implementation of CPU's LSU)
- Accesses to the same transaction unit from a single **warp** are Coalesced (i.e., LSU handles those logical accesses as a single physical access)
  - If `size_of(item)`=2^K, access to 2^{5-K} items ($=2^{5-K}$ threads) can be coalesced at most
- Memory addresses accessed by threads of **warp** (consists of 32 threads whose `Id.x` are consecutive) must be serial and dense

To reduce bottlenecks : coalescing memory access





Tiling: effective use of shared memory

Shared memory is equivalent to a **user-managed cache**

To avoid race conditions: effective use of parameters

To avoid race conditions: barrier synchronization

To avoid race conditions: Cooperative groups

(Reluctantly) resolve race conditions: Atomic accumulations

Contents

1. What is CUDA?: Introduction
2. A CUDA program for beginners
3. How CUDA works
4. Optimize CUDA program
5. Practice Problems
   - 5.1 k-NN for vector Database
   - 5.2 SW's local alignment; DP acceleration with GPU

Smith Water 's DP

Further problems ...

Appendices

(Auxiliary) Hardware implementation of LSU

```
// lsu.v
module lsu(
  input clk, rstd,              // clock
  input [2:0] func_code_in,     // function code (LB, LH, LW, LBU, LHU, SB, SH, SW)
  input [31:0] alu_result_in,   // ALU result (potential load address, store address)
  input [31:0] s_data,          // store data
  ...
  output reg [31:0] l_data,     // data loaded from memory
  ...
);
  reg [31:0] s_word;        // word to store in memory
  reg [3:0] we;             // write enable
  wire [1:0] s_tag = alu_result_in[1:0];        // tag
  wire [13:0] s_base = alu_result_in[15:2];     // base address

  ram M_data(
    .clk(clk),  // control signals
    // input
    .we(we),
    .r_addr(l_base_M),
    .w_addr(s_base),
    .w_data(s_word),
    // output
    .r_data(l_word)
  );
  ...
```

(Auxiliary) Hardware implementation of CPU's LSU

```
1  /* M-stage */
2  always @(*) begin
3    if (is_store == `ENABLE) begin // transform s_data to 4byte s_word as it corresponds to we
4      case(func_code_in)
5        `SB: begin
6          case (s_tag)
7            2'b00: begin s_word = {24'b0, s_data[7:0]};  we = 4'b0001; end
8            2'b01: begin s_word = {16'b0, s_data[7:0], 8'b0};  we = 4'b0010; end
9            2'b10: begin s_word = {8'b0, s_data[7:0], 16'b0};  we = 4'b0100; end
10           2'b11: begin s_word = {s_data[7:0], 24'b0};  we = 4'b1000; end
11           ...
12         endcase
13       end
14       `SH: begin
15         case(s_tag)
16           2'b00: begin s_word = {16'b0, s_data[15:0]}; we = 4'b0011; end
17           2'b01: begin s_word = {8'b0, s_data[15:0], 8'b0};  we = 4'b0110; end
18           2'b10: begin s_word = {s_data[15:0], 16'b0}; we = 4'b1100; end
19           ...
20         endcase
21       end
22       `SW: begin
23         s_word = s_data;  we = 4'b1111;
24       end
25       ...
26     endcase
27   end else begin s_word = 32'b0; we = 4'b0000; end
28 end
```

(Auxiliary) Hardware implementation of LSU

```
1  /* W-stage */
2  always @(*) begin
3    if (is_load_W == `ENABLE) begin
4      case(func_code_W) // transform l_word to l_data as it corresponds to func_code
5        `LB: begin
6          case (l_tag_W)
7            2'b00: l_data = {{24{l_word[7]}}, l_word[7:0]};
8            2'b01: l_data = {{24{l_word[15]}}, l_word[15:8]};
9            2'b10: l_data = {{24{l_word[23]}}, l_word[23:16]};
10           2'b11: l_data = {{24{l_word[31]}}, l_word[31:24]};
11           default: l_data = 32'b0;
12         endcase
13       end
14       `LH: begin
15         case (l_tag_W)
16           2'b00: l_data = {{16{l_word[15]}}, l_word[15:0]};
17           2'b01: l_data = {{16{l_word[23]}}, l_word[23:8]};
18           2'b10: l_data = {{16{l_word[31]}}, l_word[31:16]};
19           default: l_data = 32'b0;
20         endcase
21       end
22       `LW: begin  l_data = l_word; end // l_addr[1:0] == 2'b00 はコンパイラが保証
23       `LBU: begin  ...  end
24       `LHU: begin  ...  end
25       default l_data = 32'b0;
26     endcase
27   end else begin l_data = 32'b0; end
28 end
```