CSC4005
Parallel Programming
Tutorial 5

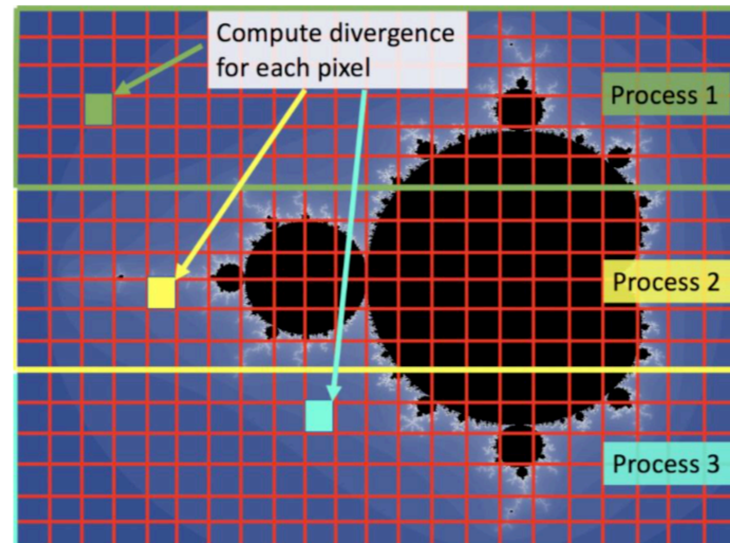
Bokai Xu, 119010355@link.cuhk.edu.cn

Outline of Tutorial 5

- **More about Project 2**
  - Distribute data points in advance (Static Scheduling)
  - Dynamically schedule jobs using minibatch (you may need a **thread pool**)
  - Modify compute function to improve performance
- **GPU & CUDA**
  - SM core
  - Multithreading
  - Memory: Registers, Local shared memory (L1 cache), L2 cache, Global memory
  - Blocks and threads
- **Utility for Writing Technical Report**

More about Project 2

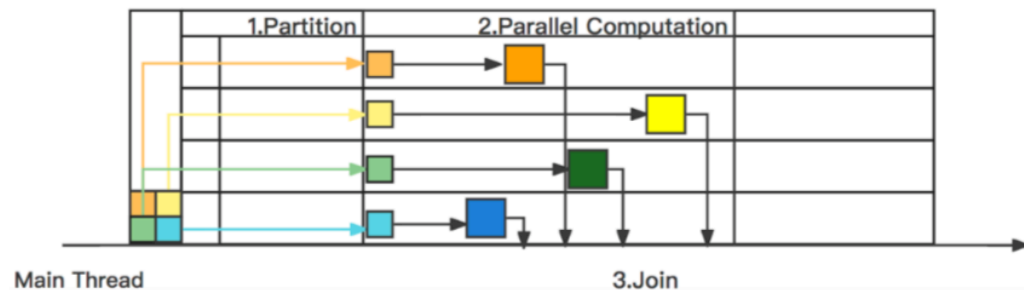
- The idea is to do data parallelization.



acknowledgement : Zhou Yutao (118010459)

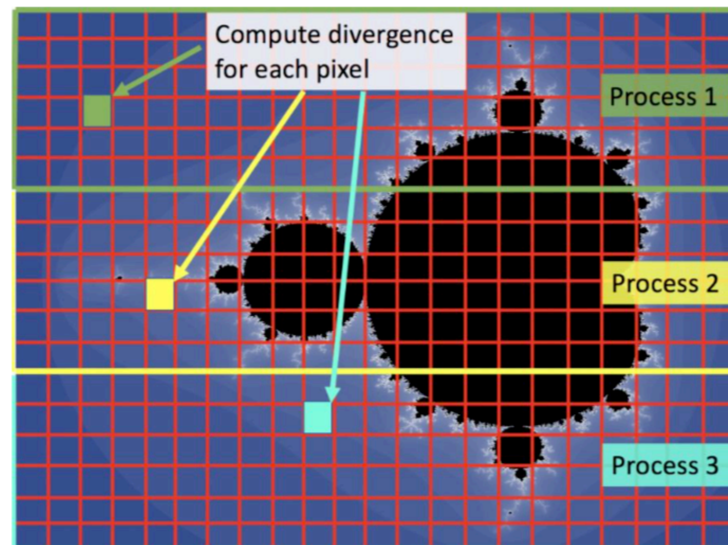
More about Project 2

- The idea is to do data parallelization.
- The basic approach is to distribute all data points to workers in advance.
- However, workers will not terminate at the same time.
- Algorithms needed.



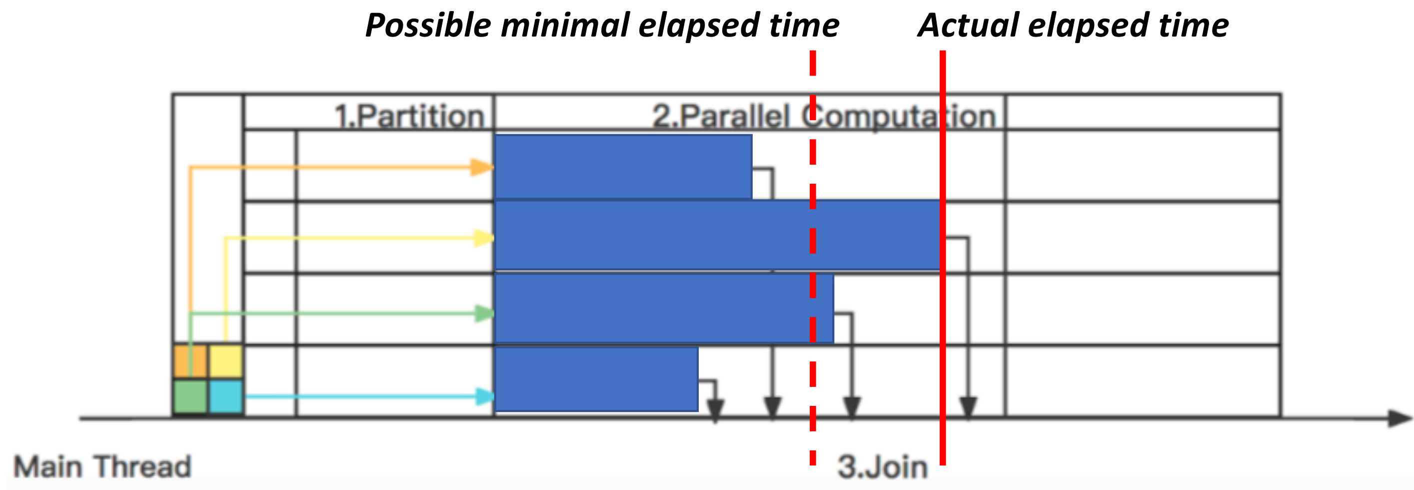
acknowledgement : Zhou Yutao (118010459)

More about Project 2



acknowledgement : Zhou Yutao (118010459)

More about Project 2



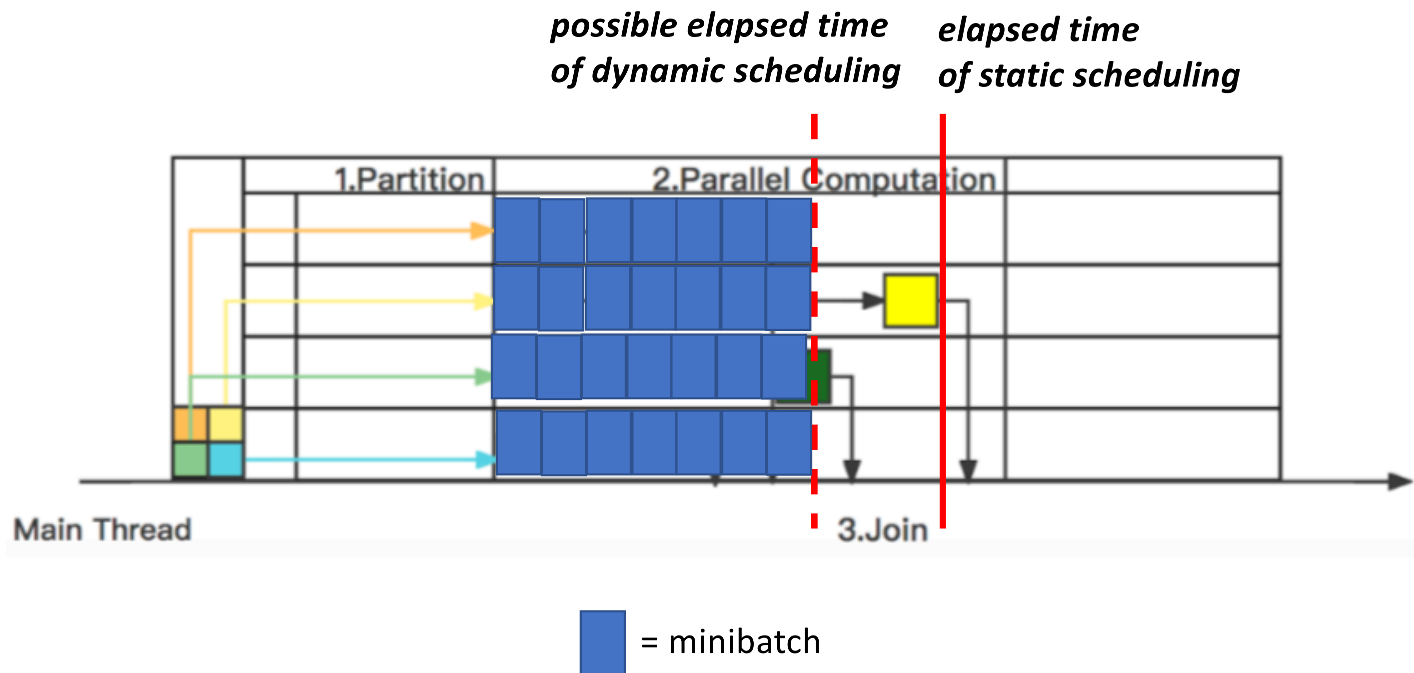
How about randomly allocate data points?

More about Project 2

- Another approach is to utilize a thread pool (fixed capacity), with which we can dynamically schedule the computational task.
- Data points are divided into many **minibatches**.
- **Batch size** and **capacity of thread pool** are possible experimental variables.

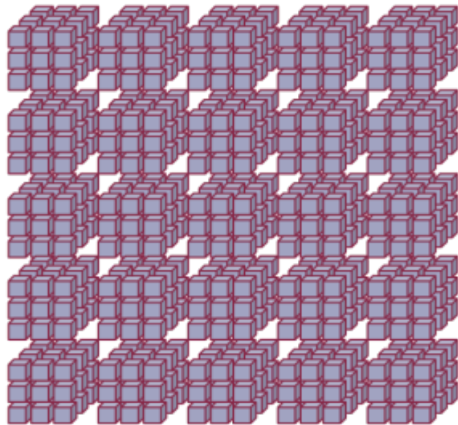
acknowledgement : Zhu Chuyan (119010486)

More about Project 2



Discussion from Tutorial 4

- Is multithreading always faster than multiprocessing?
  - **Computational intensive / I/O intensive**
- How about using **enormous amount of** degenerated CPU cores (simpler instructions, slower), which can only execute simple tasks --- **GPU**.



Up to 1024 threads can be reside on a single SM core at a specific time. But only 32 of them can be simultaneously executed.

Tesla V100 GPU has 80 SM cores.

CUDA

CUDA thread block (software concept)

- Up to **1024** CUDA threads form a **CUDA thread block**.
- Data can be distributed to **multiple CUDA blocks** (depending on experiment result).
- Each **CUDA thread block** has an unique BlockIdx.
- Each thread in a **CUDA thread block** has an unique ThreadIdx.
- We use BlockIdx and ThreadIdx to **identify** all threads.

Let’s go to hardware part.

GPU structure

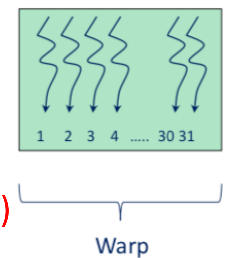
- Tesla V100 GPU has **80 SM** (streaming multiprocessor).

1 SM has **4** SM blocks (different from CUDA thread block).

Each has

- 16 FP32 Cores
- 8 FP64 Cores
- 16 INT32 Cores
- 128KB L1 / Shared memory
- 64 KB Register File

32 concurrent threads (real concurrent) can run on **1** SM (also called a **warp**)

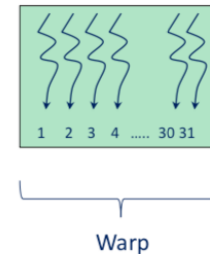


acknowledgement : Prof.Wes Armour (Oxford)

GPU design philosophy: SM (Streaming Multiprocessor)

Key feature of SM: *Single Instruction Multiple Data*.

32 threads on **1 SM** (**a warp**) execute the ***same instructions (kernel function)*** simultaneously, but with ***different data.***




```
// Kernel - Adding two matrices MatA and MatB
__global__ void MatAdd(float MatA[N][N], float MatB[N][N],
float MatC[N][N]) {
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        int j = blockIdx.y * blockDim.y + threadIdx.y;
        if (i < N && j < N) MatC[i][j] = MatA[i][j] + MatB[i][j];
}
```

Single instruction (single kernel function)
Multiple data (retrieve unique data by using index)

Wait! So what is the relationship between 32 and 1024… quite confused!

1 CUDA thread block has up to 1024 threads, and they will be divided into up to 1024/32=32 warps. All warps will be executed within the same SM.

SM

GPUs do not utilize ***cpu-style context switching, while*** GPU has its own mechanism of context switch.

```
// Kernel - Adding two matrices MatA and MatB
__global__ void MatAdd(float MatA[N][N], float MatB[N][N],
float MatC[N][N]) {
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        int j = blockIdx.y * blockDim.y + threadIdx.y;
        if (i < N && j < N) MatC[i][j] = MatA[i][j] + MatB[i][j];
}
```

reading data from memory is slower than registers

Context Switch

- When read from/write to memory, **another warp may occupy a SM** (because read/write cost $\sim 10^2$ **cycle** (processing unit survival period)).
- Execution alternates between “active” warps, with warps becoming temporarily “inactive” when waiting for data.
- A total of **1024 resident threads** (either **running (only 32)** or ready or blocking) can exist simultaneously within 1 **SM**.
- Threads in the same **warp** always execute the **same command**.

```
// Kernel - Adding two matrices MatA and MatB
__global__ void MatAdd(float MatA[N][N], float MatB[N][N],
float MatC[N][N]) {
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        int j = blockIdx.y * blockDim.y + threadIdx.y;
        if (i < N && j < N) MatC[i][j] = MatA[i][j] + MatB[i][j];
}
```

Fast Slow

acknowledgement : Prof.Wes Armour (Oxford)

SM

1 SM can execute **several concurrent CUDA thread blocks**, depending on the resources needed by all blocks.

It’s about scheduling.

GPU design philosophy: GPU Memory

- Why does a warp contain **32** threads? (registers)
- Why is the capacity of **resident threads** is **1024**? **(discussion question)**

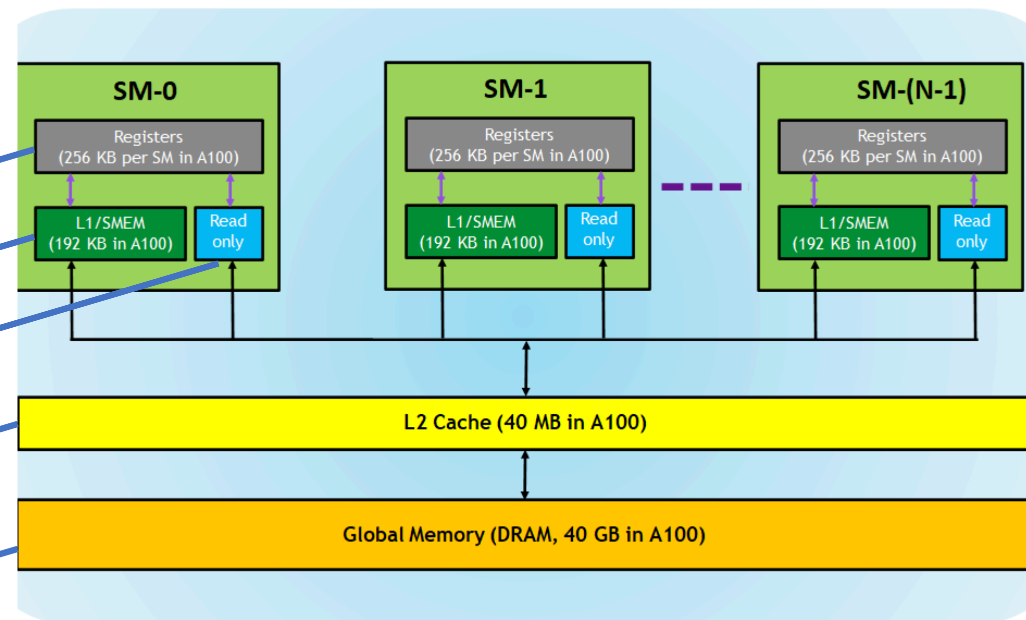
Registers are the **fastest memory** on the GPU

L1 shared memory (extremely fast)
Shared within blocks
user can allocate

Instructions

Cache for Global Memory,
faster than Global Memory

Global Memory, the slowest



GPU Memory

- **Registers**—These are private to each thread, which means that registers assigned to a thread are not visible to other threads. The compiler makes decisions about register utilization.
- **L1/Shared memory (SMEM)**—Every SM has a fast, on-chip scratchpad memory that can be used as L1 cache and shared memory. All threads in a CUDA block can share shared memory, and all CUDA blocks running on a given SM can share the physical memory resource provided by the SM..
- **Read-only memory**—Each SM has an instruction cache, constant memory, texture memory and RO cache, which is read-only to kernel code.
- **L2 cache**—The L2 cache is shared across all SMs, so every thread in every CUDA block can access this memory. The **NVIDIA A100 GPU** has increased the L2 cache size to 40 MB as compared to 6 MB in V100 GPUs.
- **Global memory**—This is the framebuffer size of the GPU and DRAM sitting in the GPU.

GPU Memory

Challenge: how many registers are we using?

Can you guess how many registers are we using in the following **vector_add** code?

```
extern "C" __global__ void vector_add(const float * A, const float * B, float * C, const int size) {
    int item = (blockIdx.x * blockDim.x) + threadIdx.x;
    if ( item < size ) {
        C[item] = A[item] + B[item];
    }
}
```

https://carpentries-incubator.github.io/lesson-gpu-programming/06-global_local_memory/index.html

GPU Memory

- If we want to make registers use more explicit in the vector_add code, we can try to rewrite it in a slightly different, but equivalent, way.

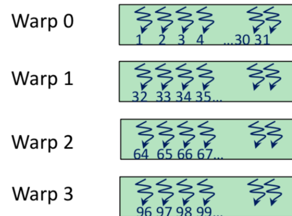
```
extern "C" __global__ void vector_add(const float * A, const float * B, float * C, const int size) {
	int item = (blockIdx.x * blockDim.x) + threadIdx.x;
	float temp_a, temp_b, temp_c;
	if ( item < size ) {
		temp_a = A[item];
		temp_b = B[item];
		temp_c = temp_a + temp_b;
		C[item] = temp_c;
	}
}
```

Hardware and Software

Software

Thread

Warp (32 threads) (minimal software scheduling unit)

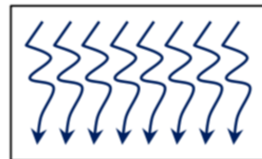


CUDA thread Block (each contains N threads) ($N < 1024$, best to be multiple of 32)

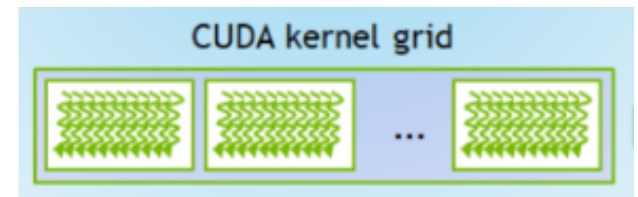
user can control

Size of Block (nthreads)

# of Blocks (nblocks)

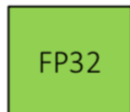


Grid (contains multiple CUDA thread blocks)

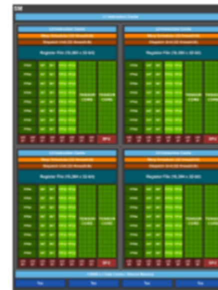


Multiple cuda thread blocks may exist on 1 SM

Hardware



CUDA core (weak compared to CPU)



SM

Device

Question: Data Partitioning

Based on this architecture, how should we partition data and allocate them to all workers?

- Use multiple-dimensional index: ***block index*** and ***thread index.***

Question: Data Partitioning

Example:

We have int x[100000] and int y[100000].

We want to compute x+y.

Let the capacity of each block be 1024 (maximum).

Number of blocks = 100000 / 1024 + 1 = 98.

For each thread, it only compute x[i]+y[i], where i=block_id * 1024 + thread_id.

CUDA Variables

Variable	Description	Property
gridDim	Total number of blocks	
blockDim	Total number of threads in a block	
blockIdx	Block id (x dimensional) of this thread	Threads in a specific block will get the same ***blockIdx***
threadIdx	Thread id (x dimensional) of this thread in its block	From 0 to 1023. Only valid within a block.

Example: Vector addition: launch a kernel

1. Declare kernel function

```
// CUDA kernel. Each thread takes care of one element of c
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get our global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;

    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}
```

Example: Vector addition: launch a kernel

2. Allocate memory in host first

```
int main( int argc, char* argv[] )
{
    // Size of vectors
    int n = 100000;

    // Host input vectors
    double *h_a;
    double *h_b;
    //Host output vector
    double *h_c;

    // Device input vectors
    double *d_a;
    double *d_b;
    //Device output vector
    double *d_c;

    // Size, in bytes, of each vector
    size_t bytes = n*sizeof(double);

    // Allocate memory for each vector on host
    h_a = (double*)malloc(bytes);
    h_b = (double*)malloc(bytes);
    h_c = (double*)malloc(bytes);
```

Example: Vector addition: launch a kernel

3. Initialize data and copy data to device (gpu) global memory

```
// Allocate memory for each vector on GPU
cudaMalloc(&d_a, bytes);
cudaMalloc(&d_b, bytes);
cudaMalloc(&d_c, bytes);

int i;
// Initialize vectors on host
for( i = 0; i < n; i++ ) {
    h_a[i] = sin(i)*sin(i);
    h_b[i] = cos(i)*cos(i);
}

// Copy host vectors to device
cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);
```

Example: Vector addition: launch a kernel

4. Determine block size and grid size.

```
int blockSize, gridSize;

// Number of threads in each thread block
blockSize = 1024;

// Number of thread blocks in grid
gridSize = (int)ceil((float)n/blockSize);
```

Example: Vector addition: launch a kernel

5. Launch the kernel function

```
// Execute the kernel
vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

// Copy array back to host
cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );
```

Example: Vector addition: launch a kernel

6. Copy result from device memory back to host memory.

```
// Execute the kernel
vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

// Copy array back to host
cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );
```

Example: Vector addition: launch a kernel

7. Release device memory and host memory.

```
// Sum up vector c and print result divided by n, this should equal 1 within error
double sum = 0;
for(i=0; i<n; i++)
    sum += h_c[i];
printf("final result: %f\n", sum/n);

// Release device memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

// Release host memory
free(h_a);
free(h_b);
free(h_c);

return 0;
```

Example: Vector addition: launch a kernel

Compile:

nvcc vec_add.cu

Run (please use slurm)

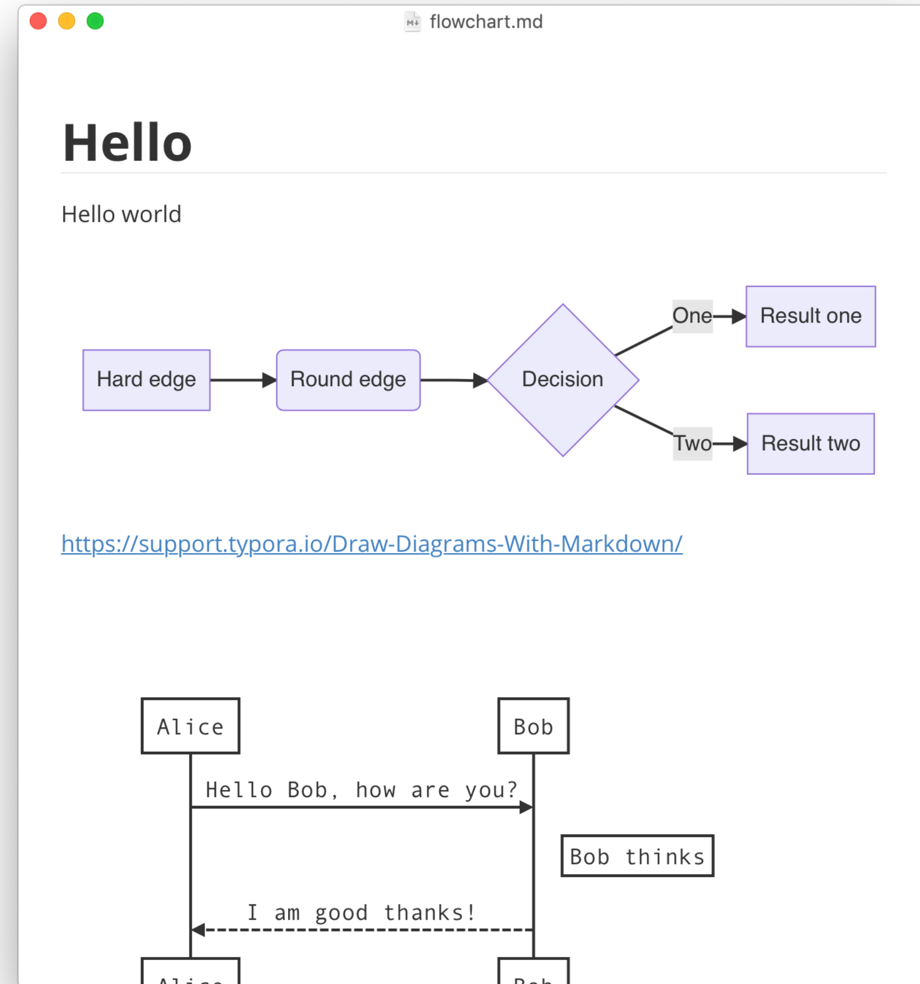
salloc -n1 -c1 --gres=gpu:1
srun ./a.out

Thank you!

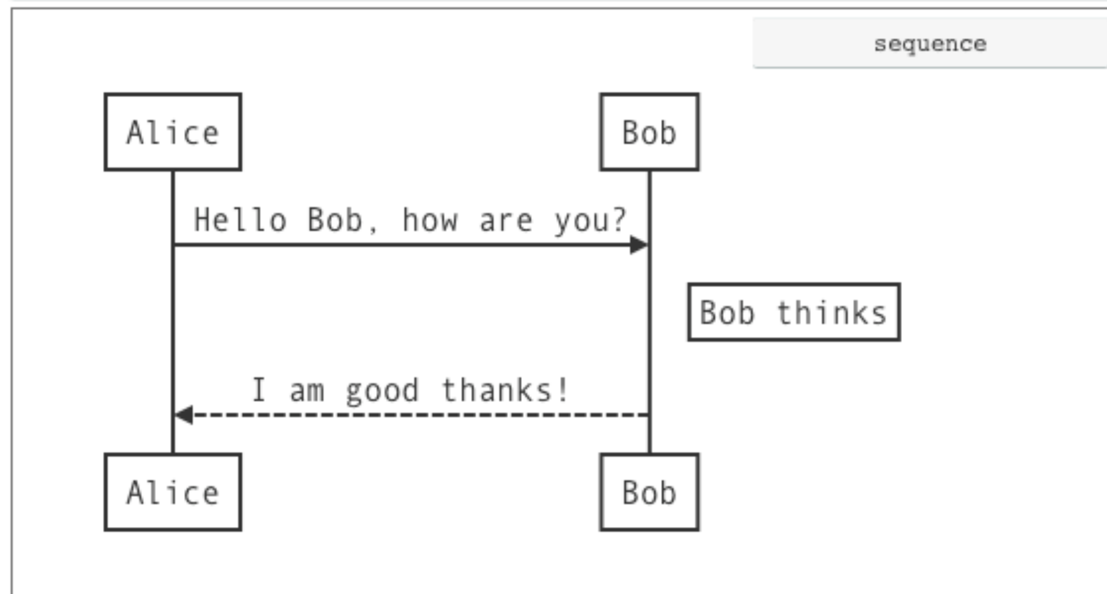
We will talk more about CUDA in the following weeks.

About report

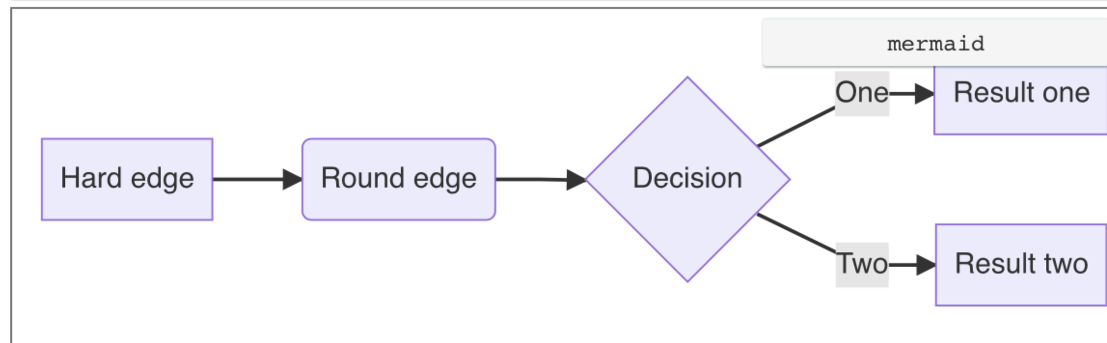
- Integrate flowchart with markdown: **Typora**



```
Alice->Bob: Hello Bob, how are you?
Note right of Bob: Bob thinks
Bob-->Alice: I am good thanks!
```



```
graph LR
A[Hard edge] -->B(Round edge)
    B --> C{Decision}
    C -->|One| D[Result one]
    C -->|Two| E[Result two]
```



- https://support.typora.io/Draw-Diagrams-With-Markdown/