Getting Start with GPU Computing: What, Why&How

Tianmu Xin

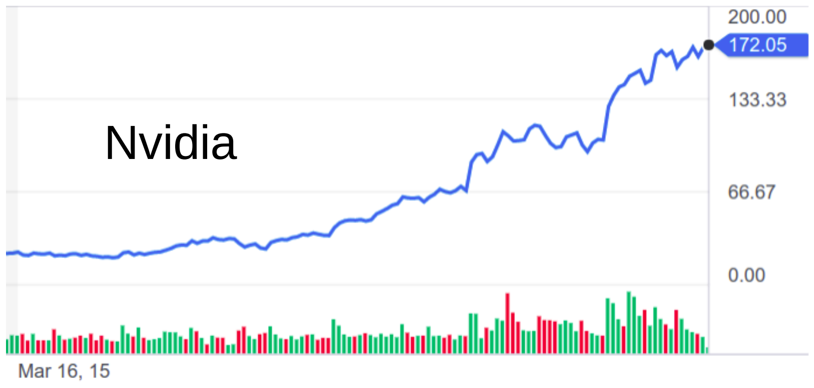
CASE Seminar, 09142017

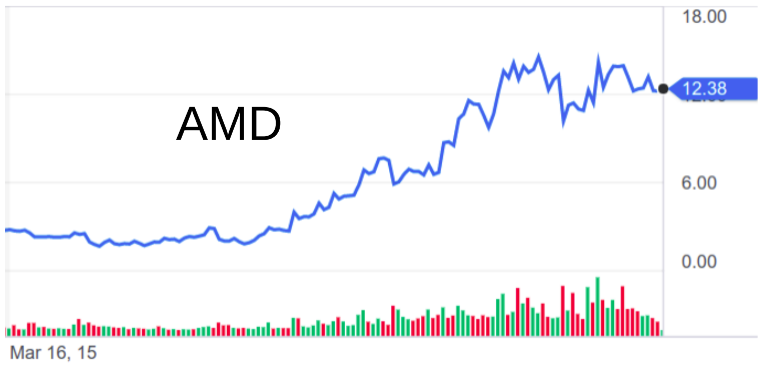
Content

- What is GPU?
- What is it good at?
- GPU computing basics
  - Architecture
  - Advantage.
  - Disadvantage.
  - How to use it?
  - Simple examples.

What is GPU?

- Graphic Processing Unit (GPU).

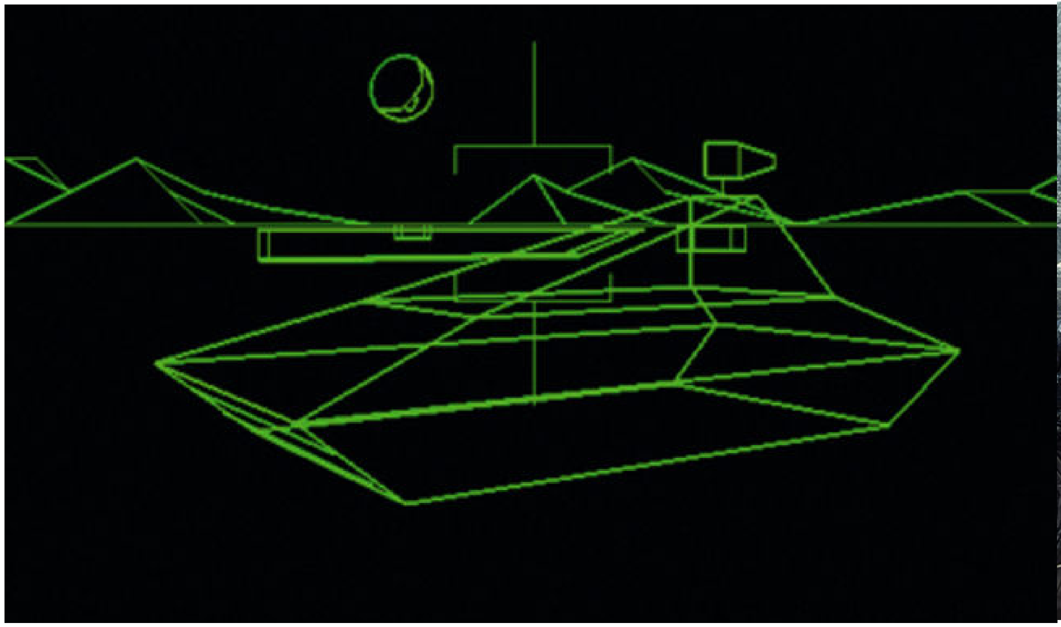




What is GPU good at?

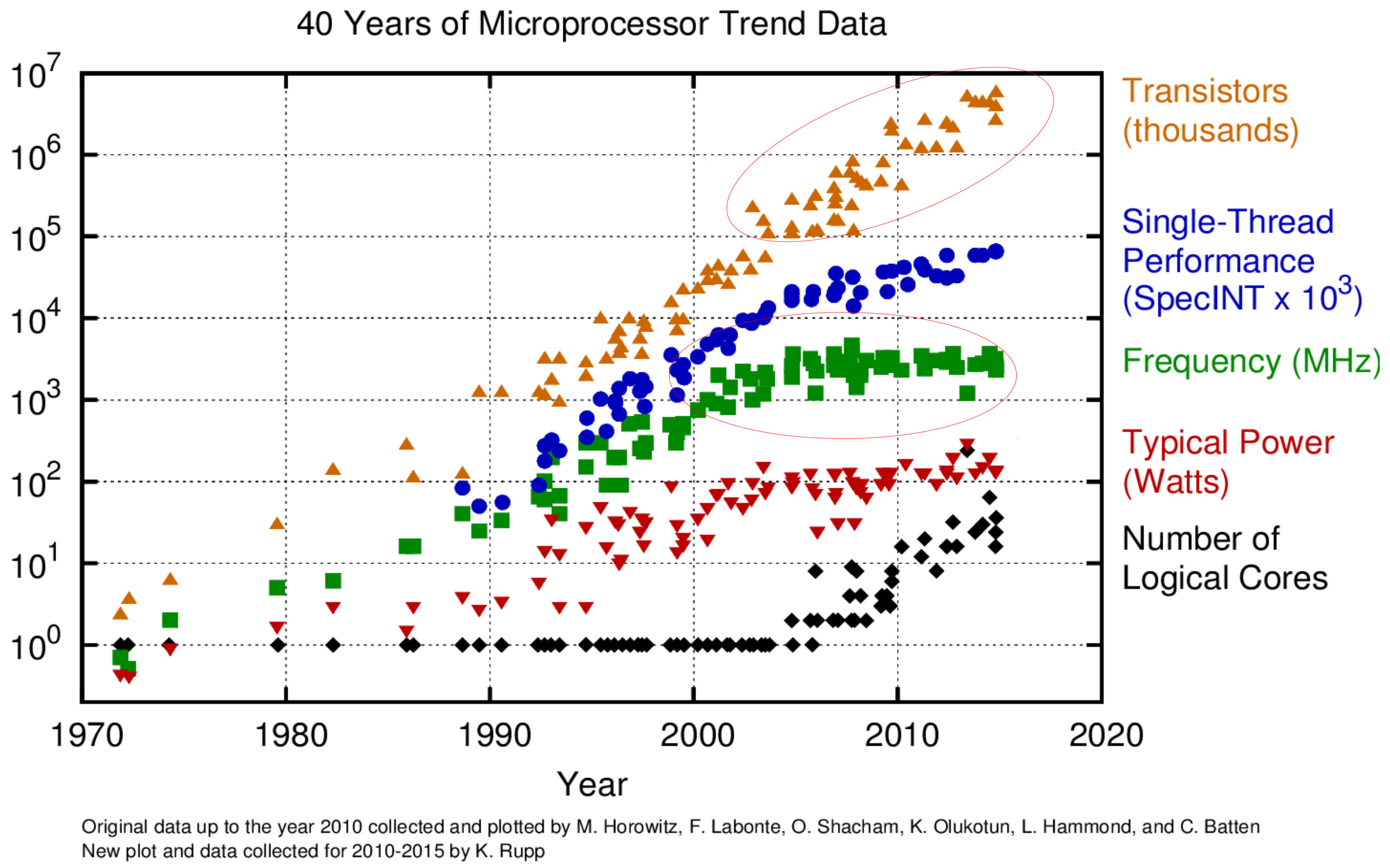
- 3D video game: Large number of vertices need to be updated.

View of point rotation = Matrix multiplication.



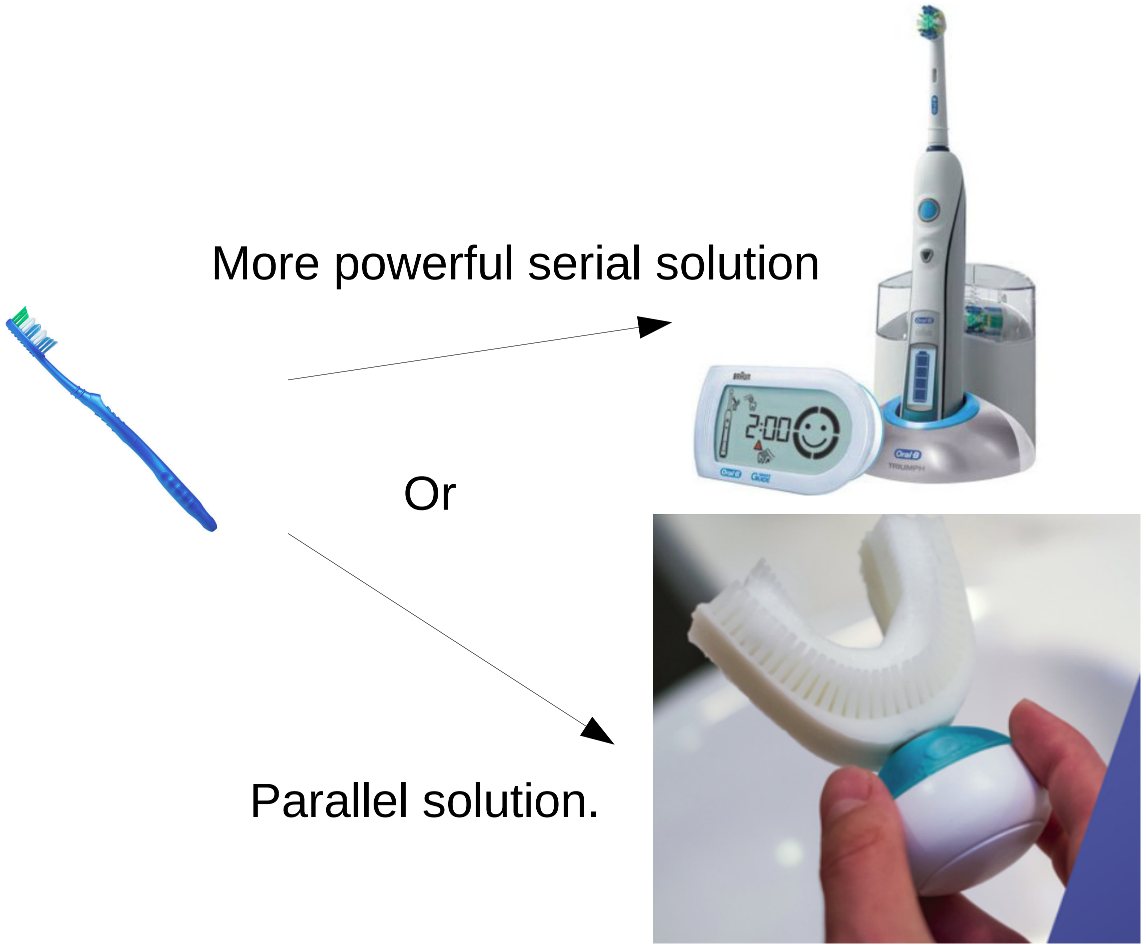
What is GPU good at?

- Parallel computing.



What is GPU good at?

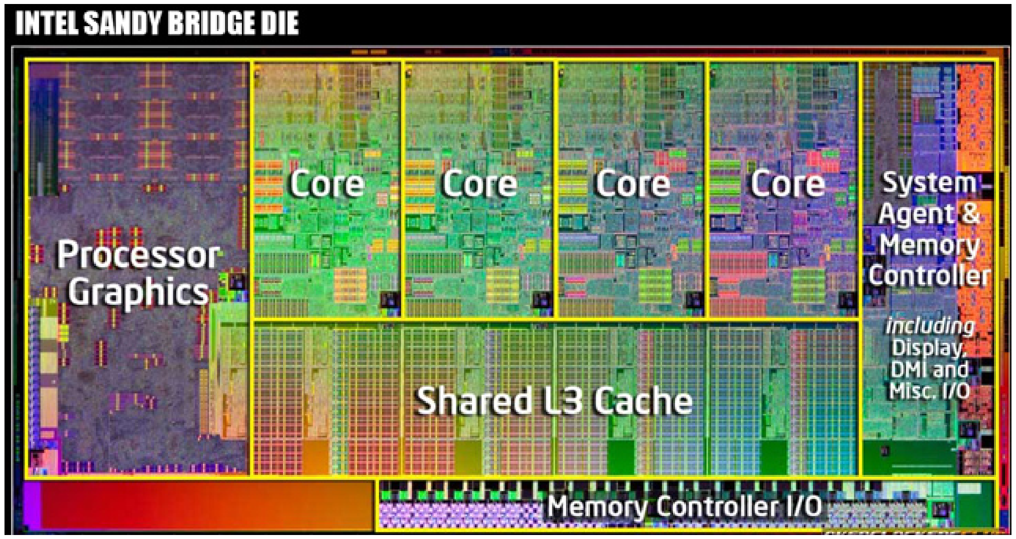
- Parallelization is the direction.
  - OpenMP
  - MPI
  - OpenCL
  - CUDA



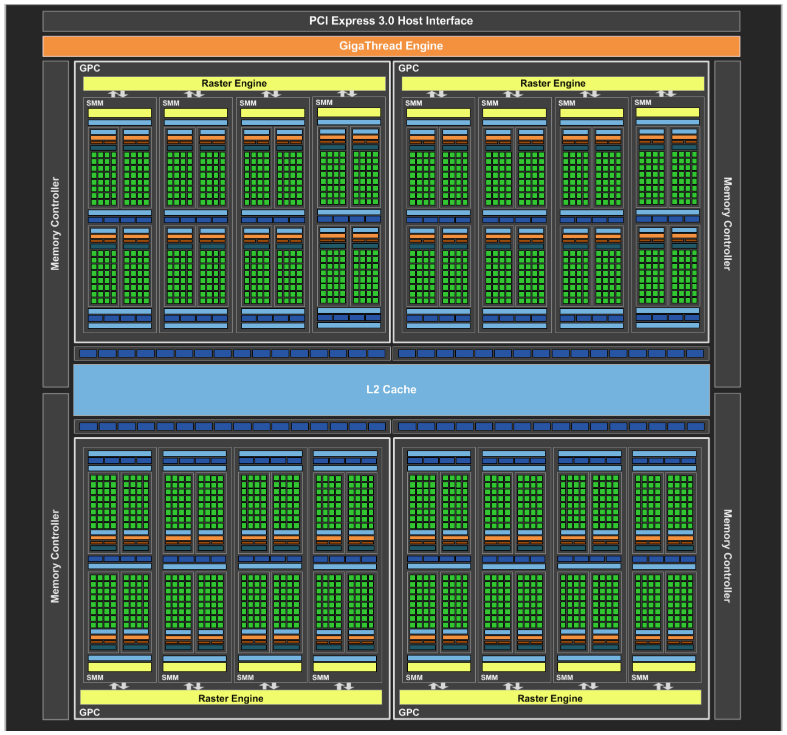
GPU Computing Basics

- Architecture.

CPU (Intel Sandy Bridge)



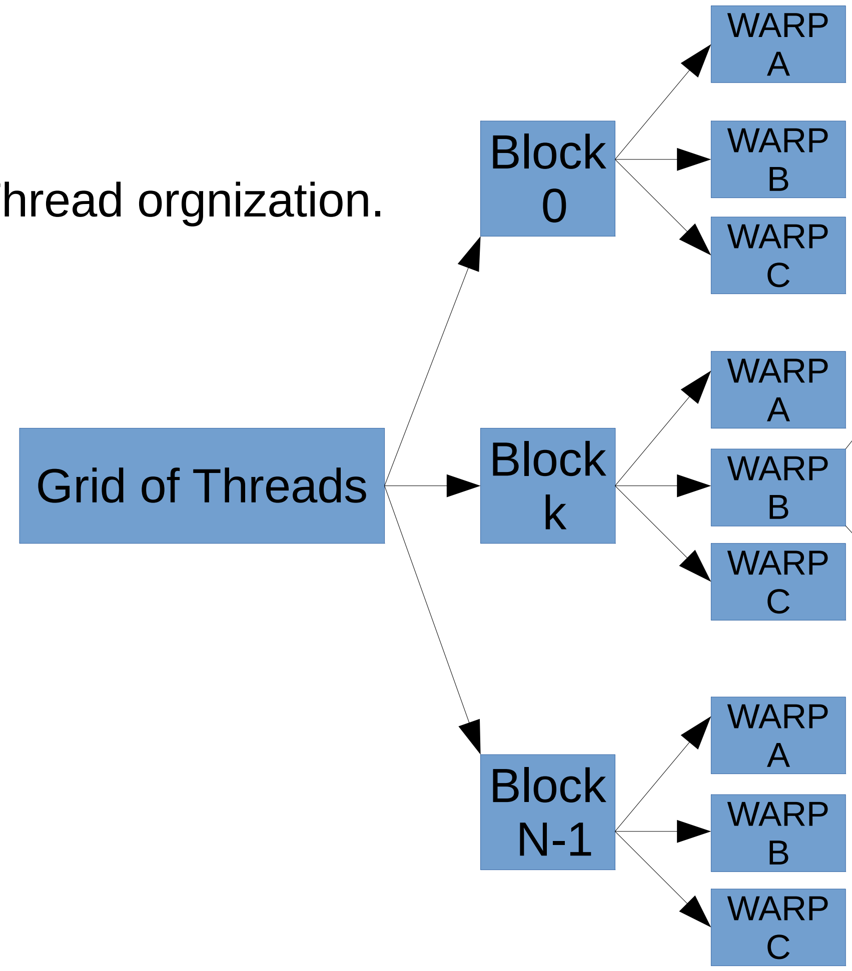
Nvidia (GM 107 Maxwell)



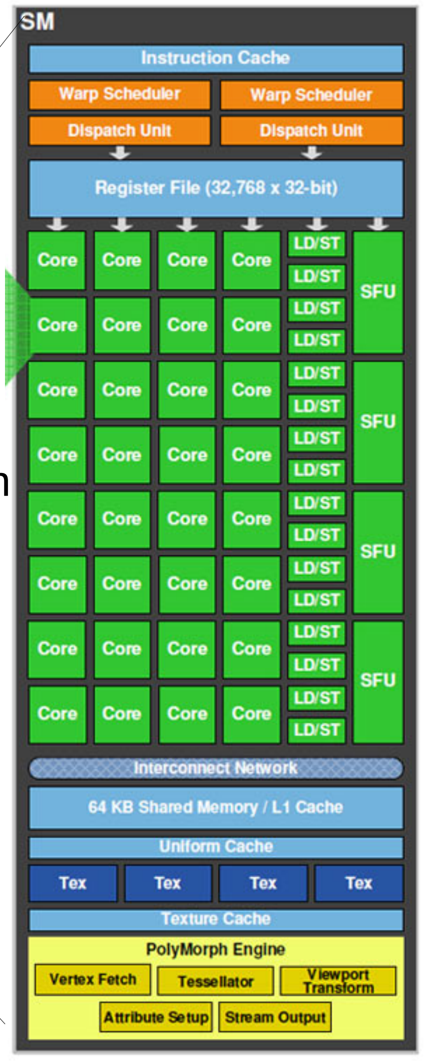
Able to run one thread with each core simultaneously. Namely 1024 threads at the same time for this chip.

GPU Computing Basics

Thread orgnization.



Same instruction for all cores on one streaming multiprocessor.



Streaming Multiprocessor (SM)

GPU Computing Basics

- Advantage.
  - Single Instruction Multiple Data (SIMD)

  e.g. Multiplication of Matrices (Embarrassingly Parallel)

$$C = A \times B$$

A, B, C are nxn matrices.

$$C_{ij} = \sum_{m=1}^{n} A_{im} B_{mj}$$

To get each elements in C, the instruction is simple:
n multiplications and n-1 adds.

GPU Computing Basics

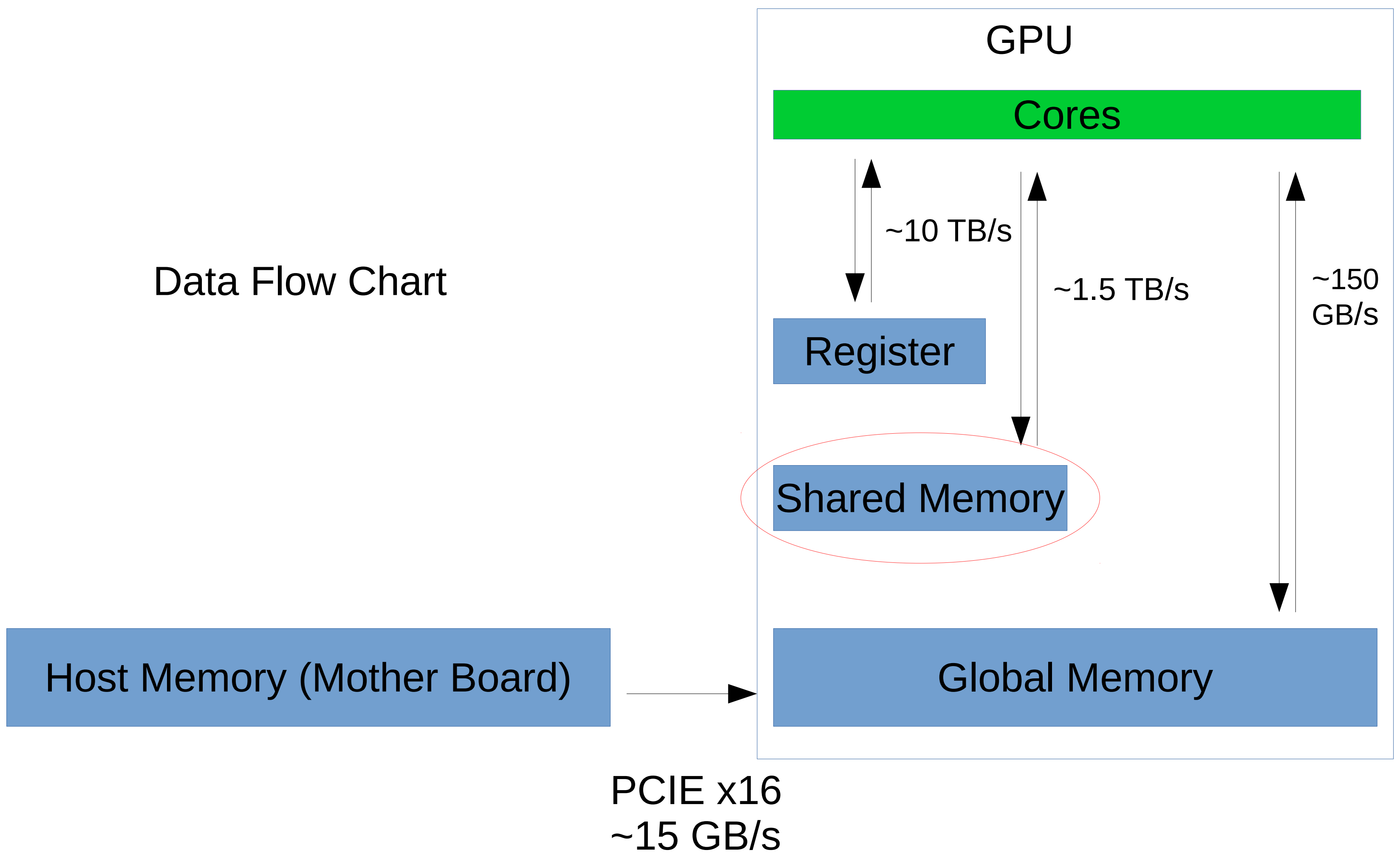
- Disadvantage
  - SIMD

    All threads in one SM (32 threads) execute the same instruction, Branching is bad.

    e.g. if statement, case switching.

How to use it?

Data Flow Chart



Example 1

- Typical work flow.

  (Matrix multiplication.)

  – Allocate,initialize memory blocks on host;

```
// allocate host memory for matrices A and B
float *h_A = (float *)malloc(mem_size_A);
float *h_B = (float *)malloc(mem_size_B);
```

  – Allocate memory blocks on device (GPU);

```
// allocate device memory
float *d_A, *d_B, *d_C;
checkCudaErrors(cudaMalloc((void **)&d_A, mem_size_A));
checkCudaErrors(cudaMalloc((void **)&d_B, mem_size_B));
checkCudaErrors(cudaMalloc((void **)&d_C, mem_size_C));
```

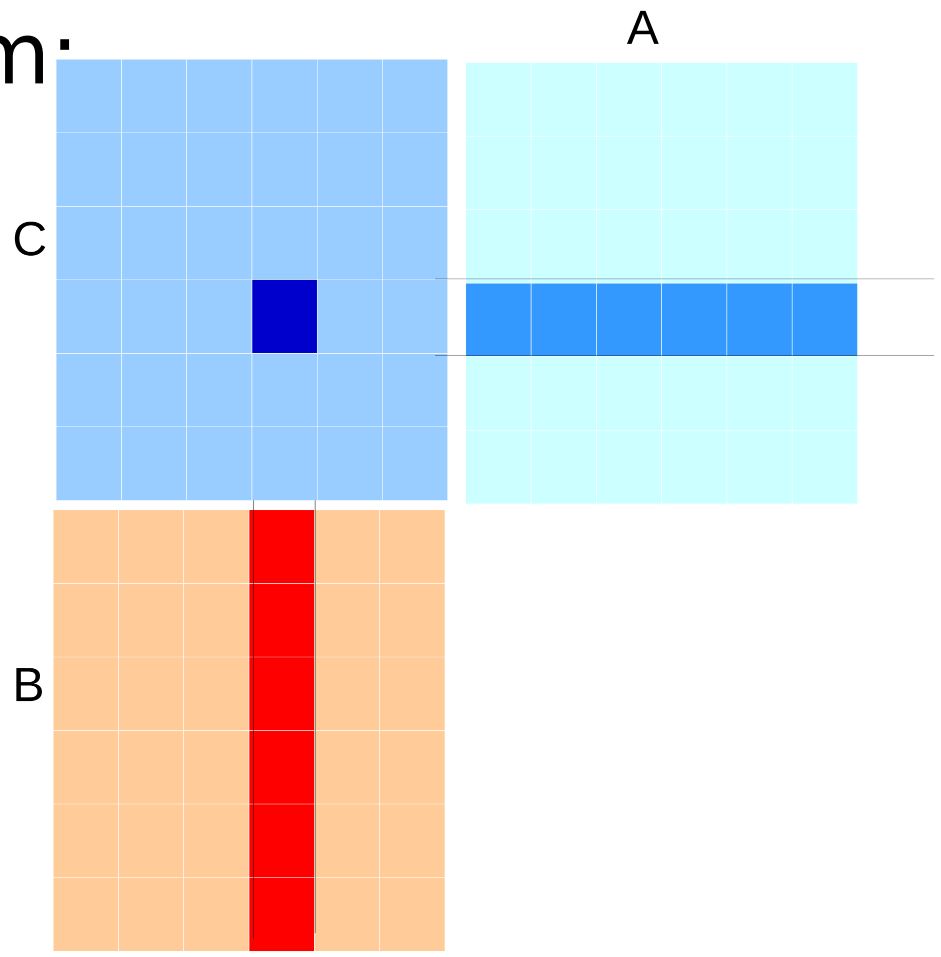
  – Transfer host memory to device memory;

```
// copy host mem to device mem
checkCudaErrors(cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice));
checkCudaErrors(cudaMemcpy(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice));
```

Example 1

Matrix multiplication.(continued)

Simple algorithm:



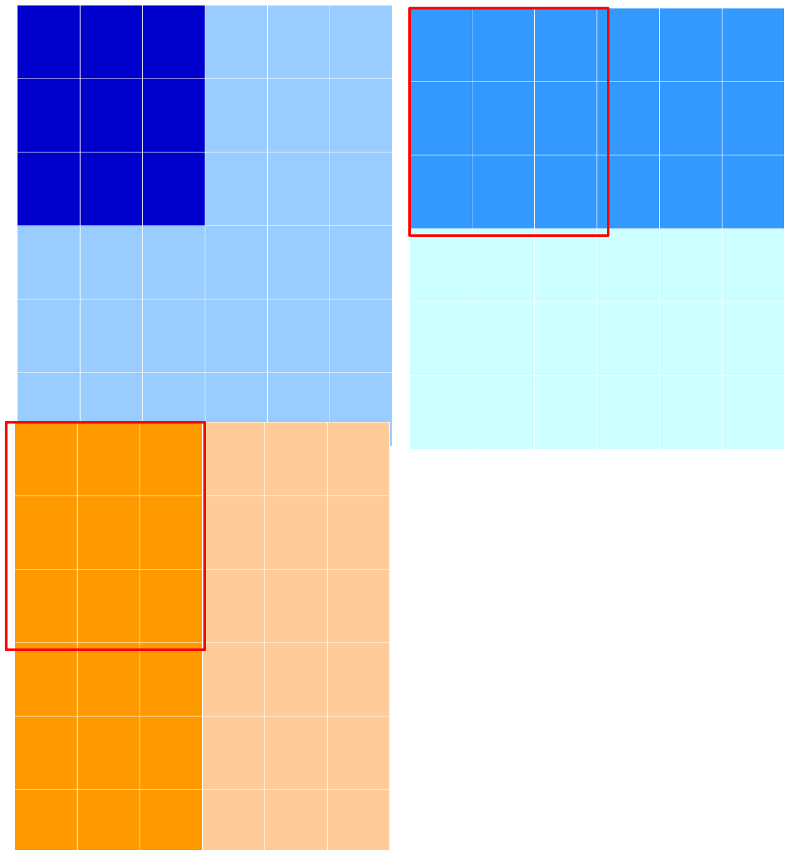
One thread calculates one element in matrix C, fetch data from global memory directly. Problem: Lots of redundant memory read.

Example 1

Matrix multiplication.(continued)

Take advantage of shared memory:

Load submatrices of A and B into shared memory, reuse the data to calculate all elements in submatrices of C.

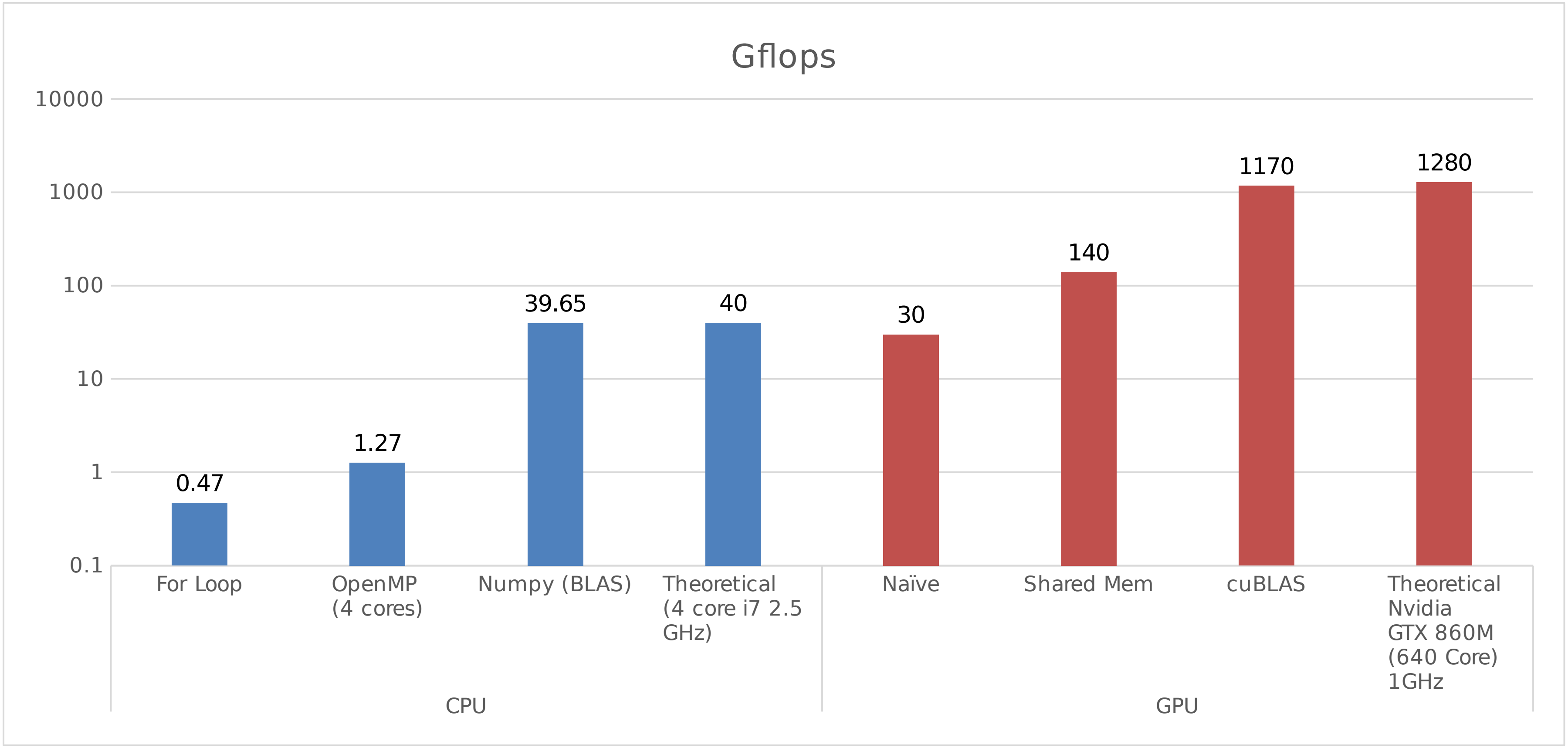


Example 1

```
        // Loop over all the sub-matrices of A and B
        // required to compute the block sub-matrix
        for (int a = aBegin, b = bBegin;
                a <= aEnd;
                a += aStep, b += bStep)
        {

                // Declaration of the shared memory array As used to
                // store the sub-matrix of A
                __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

                // Declaration of the shared memory array Bs used to
                // store the sub-matrix of B
                __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

                // Load the matrices from device memory
                // to shared memory; each thread loads
                // one element of each matrix
                As[ty][tx] = A[a + wA * ty + tx];
                Bs[ty][tx] = B[b + wB * ty + tx];

                // Synchronize to make sure the matrices are loaded
                __syncthreads();

                // Multiply the two matrices together;
                // each thread computes one element
                // of the block sub-matrix
#pragma unroll

                for (int k = 0; k < BLOCK_SIZE; ++k)
                {
                        Csub += As[ty][k] * Bs[k][tx];
                }

                // Synchronize to make sure that the preceding
                // computation is done before loading two new
                // sub-matrices of A and B in the next iteration
                __syncthreads();
        }

        // Write the block sub-matrix to device memory;
        // each thread writes one element
        int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
        C[c + wB * ty + tx] = Csub;
```

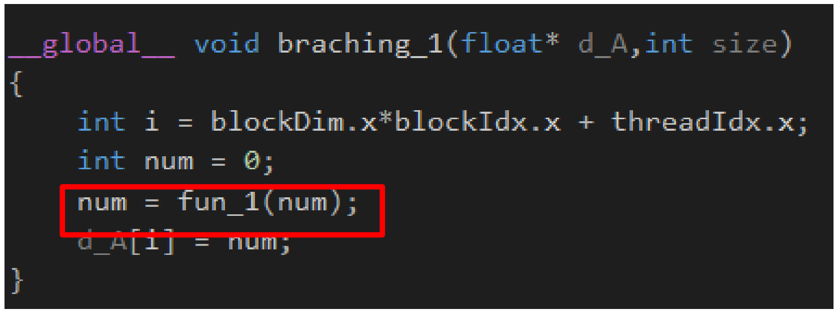
Example 1

- Performance Comparison (single precision float)



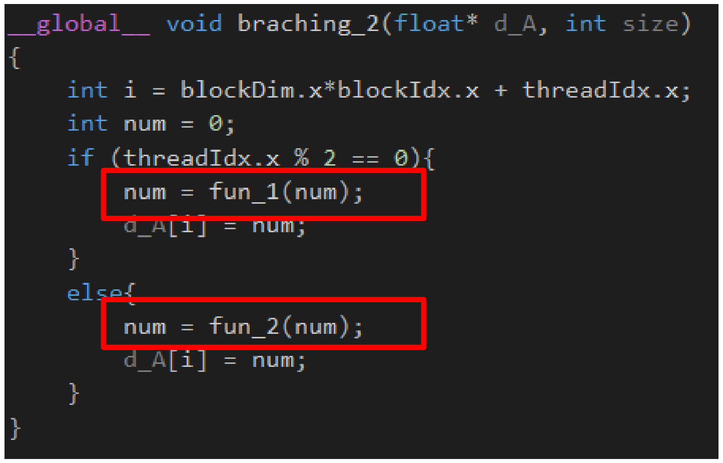
Example 2
Branching is bad

- No Branching:



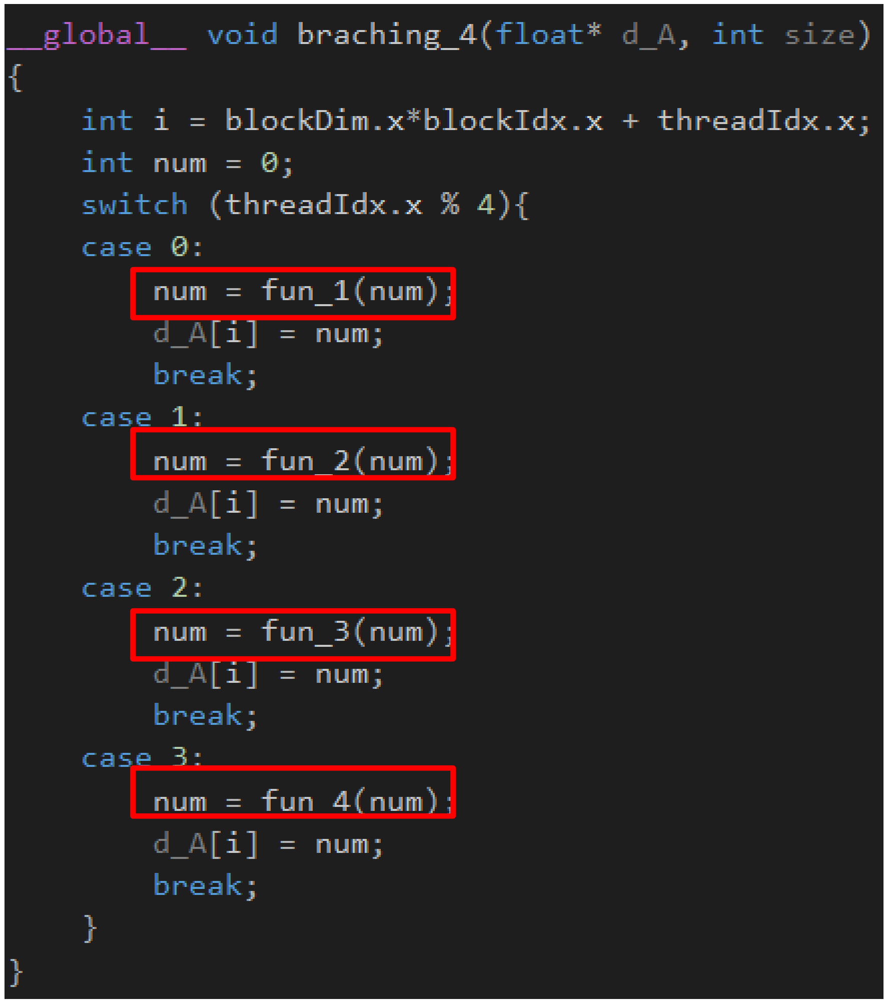
```
__global__ void braching_1(float* d_A,int size)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    int num = 0;
    num = fun_1(num);
    d_A[i] = num;
}
```

- Two Branches:



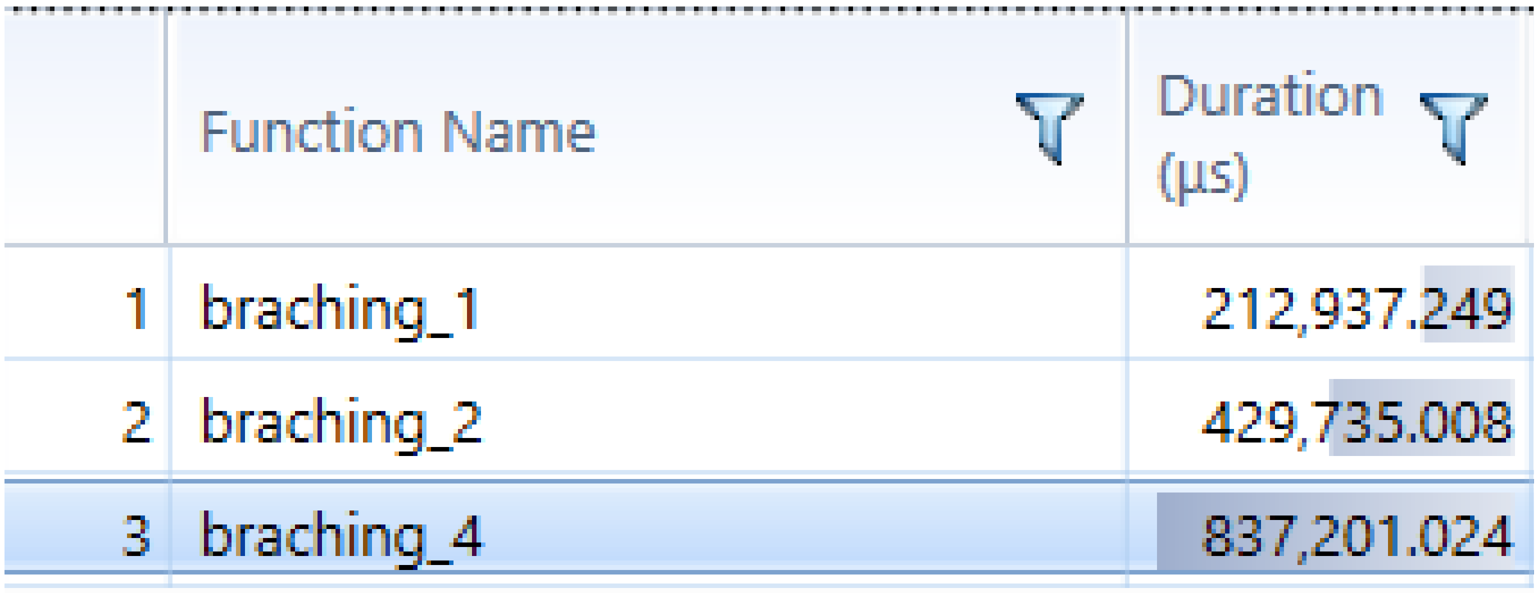
```
__global__ void braching_2(float* d_A, int size)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    int num = 0;
    if (threadIdx.x % 2 == 0){
        num = fun_1(num);
        d_A[i] = num;
    }
    else{
        num = fun_2(num);
        d_A[i] = num;
    }
}
```

- Four Branches:



```
__global__ void braching_4(float* d_A, int size)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    int num = 0;
    switch (threadIdx.x % 4){
    case 0:
        num = fun_1(num);
        d_A[i] = num;
        break;
    case 1:
        num = fun_2(num);
        d_A[i] = num;
        break;
    case 2:
        num = fun_3(num);
        d_A[i] = num;
        break;
    case 3:
        num = fun_4(num);
        d_A[i] = num;
        break;
    }
}
```

Example 2



	Function Name	Duration (µs)
1	braching_1	212,937.249
2	braching_2	429,735.008
3	braching_4	837,201.024

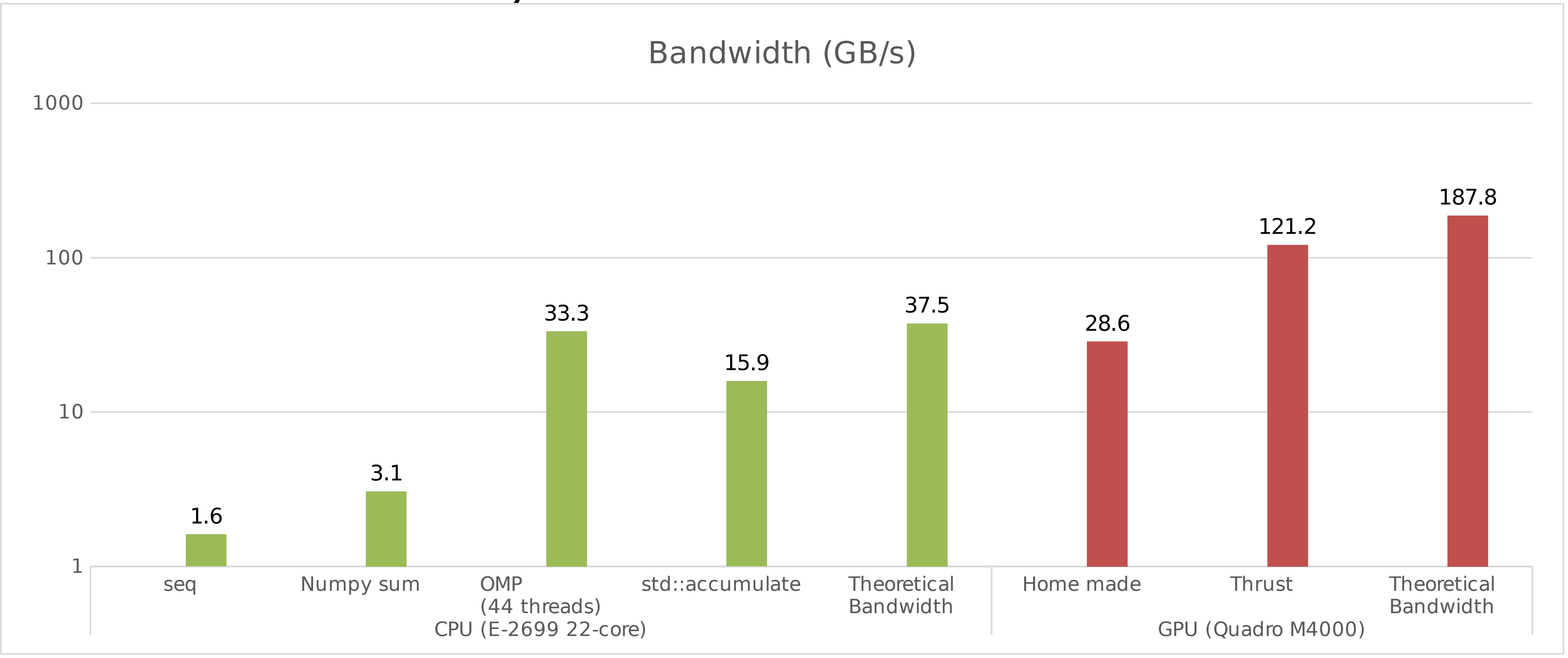
Example 3

- Reduction of array:

Array Size = 256 Million elements,

Data Type = single precision float.

Limitation: Memory Bandwidth.



Example 4

CUDA_OpenGL Interop

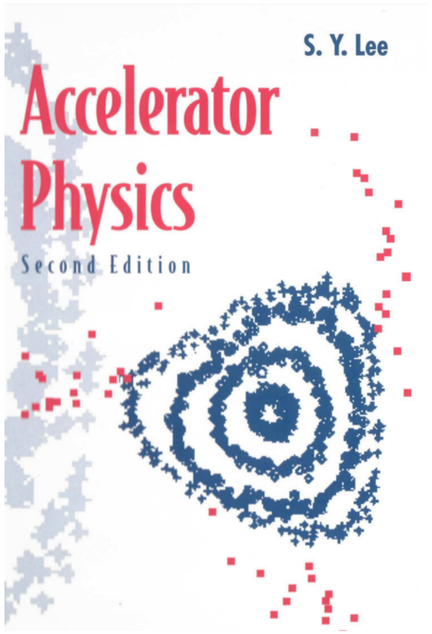
3rd order resonance driven by sextuple in ring.

- Simple transport matrix

$$\begin{pmatrix} \cos(\Phi)+\alpha\sin(\Phi) & \beta\sin(\Phi) \\ -\gamma\sin(\Phi) & \cos(\Phi)-\alpha\sin(\Phi) \end{pmatrix}$$

- Sextuple.

$$\Delta x' = -\frac{1}{2} S(x^2 - y^2),$$
$$\Delta y' = S x y$$



Run App.

Summary

- 10 times more Gflops/watts compare to CPU in some problems.
- GPU is good at solving problems which can by decomposed into small SIMD sub routines.
- Memory bandwidth is still the major bottle neck, use shared memory if possible.
- Use Library when available.
- Use Library when available.
- Use Library when available.