From CPU to GPU I

- We have discussed that improving the memory access greatly reduce the running time of matrix-matrix multiplications.
- The question now is why CPU is not efficient enough and people often use GPU.
- The short answer is that GPU can do better parallel computation than CPU on matrix-matrix multiplications.
- That is, CPU and GPU have different designs.
- However, we also notice that their designs are evolving.

From CPU to GPU II

- Some time ago, an influential paper (Fatahalian et al., 2004) showed that GPU for matrix-matrix multiplications was not faster than an optimized implementation on CPU.
- Apparently, the GPU architecture has improved a lot since then.
- Therefore, in the slides we aim to give a high-level overview instead of getting into details that may change over time.

CPU Versus GPU I

- CPU is designed to handle different types of tasks.
- Thus, in every personal computer you need a CPU.
- In contrast, GPU is more specialized.
- From the name, Graphical Processing Unit, we see that it was initially designed for graphics rendering.
- It happens that GPU is very efficient for matrix operations in deep learning.
- Specifically, it can run the same operation in a massively parallel way.

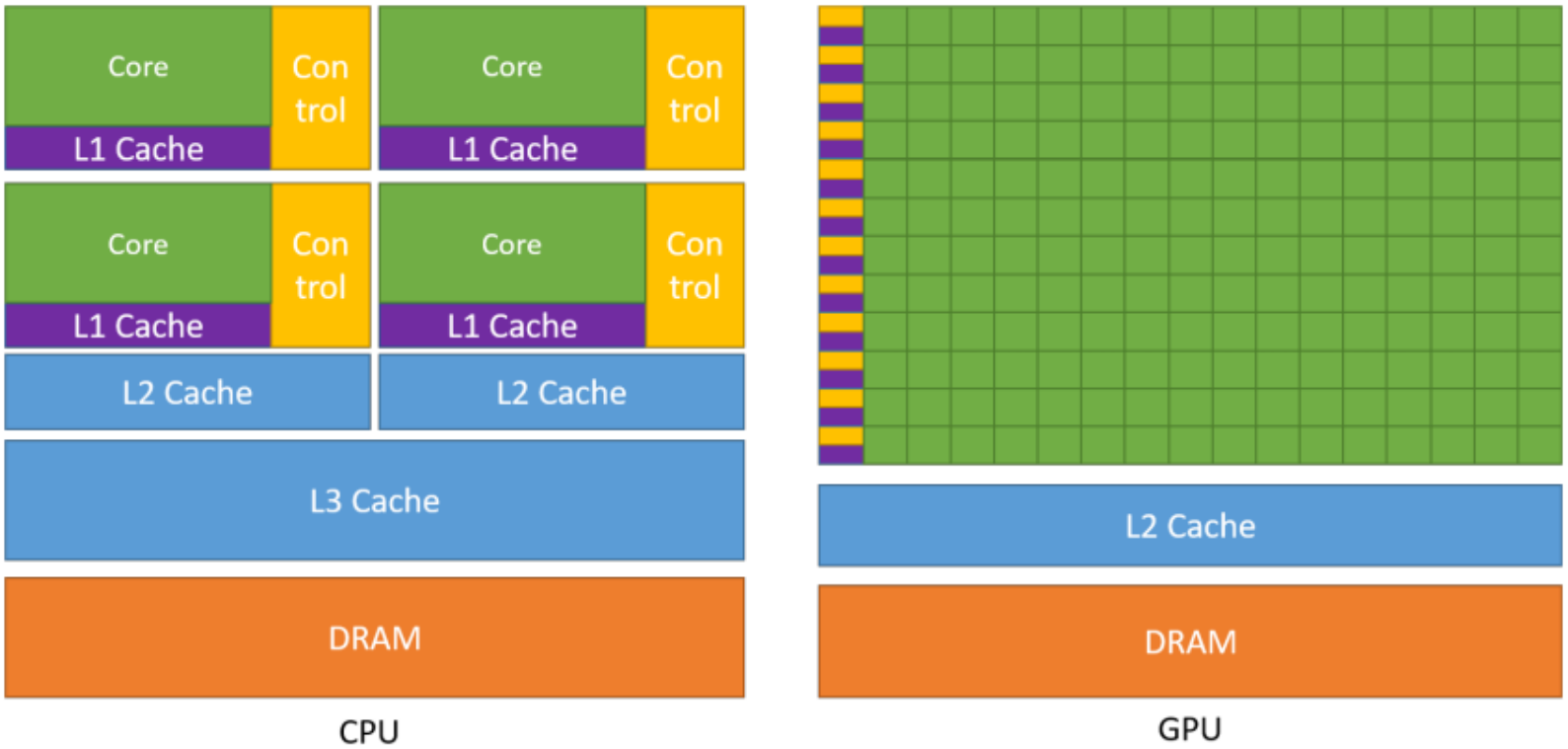
CPU Versus GPU II

- For example, if we would like to do a vector addition:

$$c_i = a_i + b_i, \forall i,$$

- We have the same "+" operation on multiple data (i.e., $a_i, b_i, \forall i$).
- The following figure from Nvidia's CUDA C++ programming guide[1] illustrates the architectural difference between CPU and GPU.

CPU Versus GPU III



CPU Versus GPU IV

- In the figure we see those GPU cores are in the green color, and the shared memory for the cores in each row is in the purple color.
- Clearly, each GPU has a lot more cores than CPU.
- From the guide, "Driven by the insatiable market demand for realtime, high-definition 3D graphics, GPU has evolved into a highly parallel, multithreaded, manycore processor with tremendous computational horsepower and very high memory bandwidth."

CPU Versus GPU V

- To utilize the so many cores, it is convenient if we split them to several groups.
- We do this in both software level and hardware implementation.
- The separation is important as the hardware implementation may rapidly evolve.
- Thus, we have some kind of abstraction, so users only need to focus on the software part.
- We call the software solution a GPU programming model.

[1] https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

GPU Programming Model I

- We follow the termology in CUDA programming guide though concepts in other GPU programming models are similar.
- In CUDA, we achieve parallelism by running many threads. Each thread is executed on a GPU core.
- A typical CPU/GPU core has multiple threads. For example, x86 CPU processors have two threads per core.

GPU Programming Model II

- A thread and a core are the most basic unit on the software and hardware sides, respectively. For simplicity, let us assume that they correspond to each other.
- To handle many cores, CUDA considers a hierarchy of thread groups.
- That is, in a higher level, we have coarse-grained data parallelism and task parallelism.
- In the lower level, we have fine-grained data parallelism and thread parallelism.
- Specifically,

GPU Programming Model III

a grid contains blocks

$\downarrow$

a block contains threads

- In CUDA, a kernel is a function to be executed in parallel.
- It may involve several threads, or several blocks of threads (i.e., a grid).
- Let us consider two examples in CUDA programming guide.
- In the first example, a kernel invokes N threads and each thread does one pair-wise addition.

GPU Programming Model IV

```
__global__ void VecAdd(float* A, float* B,
                       float* C)
{
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}
int main()
{
  ...
  // Kernel invocation with N threads
  VecAdd<<<1, N>>>(A, B, C);
```

GPU Programming Model V

```
    ...
}
```

- `__global__` indicates that this is a kernel function that can be called from the host side (i.e., `main()` on CPU) by using `<<<...>>>`.
- `<<<1, N>>>` specifies the grid and block sizes. More examples will be given later.
- Here we have a grid with one block, and each block has N threads.

GPU Programming Model VI

- A block can be a one-dimensional, two-dimensional, or three-dimensional block of threads. In our example, the block is one-dimensional.
- Such a design is natural. We see that a two-dimensional block of threads can easily handle matrix operations.
- We can further use a grid of equally-shaped thread blocks to run a kernel.
- Similarly, a grid may contain one-dimensional, two-dimensional, or three-dimensional blocks.

GPU Programming Model VII

- The next example shows the use of a grid of blocks for matrix addition.

```
__global__ void MatAdd(float A[N][N],
float B[N][N], float C[N][N])
{
    int i = blockIdx.x * blockDim.x +
            threadIdx.x;
    int j = blockIdx.y * blockDim.y +
            threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
```

GPU Programming Model VIII

```
}

int main()
{
    ...
    int block_size = 16;
    dim3 dim_block(block_size, block_size);
    dim3 dim_grid(N/block_size,
       N/block_size);
    MatAdd<<<dim_grid, dim_block>>>(A,
       B, C);
```

GPU Programming Model IX

```
    ...
}
```

- CUDA has a special data type `dim3` to specify the dimensionality of a block and a grid.
- Here each our block has

$$16 \times 16 = 256$$

threads.

GPU Programming Model X

- Then we need a two-dimensional setting of

$$\frac{N}{16} \times \frac{N}{16} \text{ blocks,}$$

  where our matrices are $N \times N$.
- For simplicity, we assume that N is divisible by 16.
- Note that we do not specify the third dimension of `dim3`. Or we can say that the third dimension is 1.
- To access the row index, we see the following line

```
int i = blockIdx.x * blockDim.x +
threadIdx.x;
```

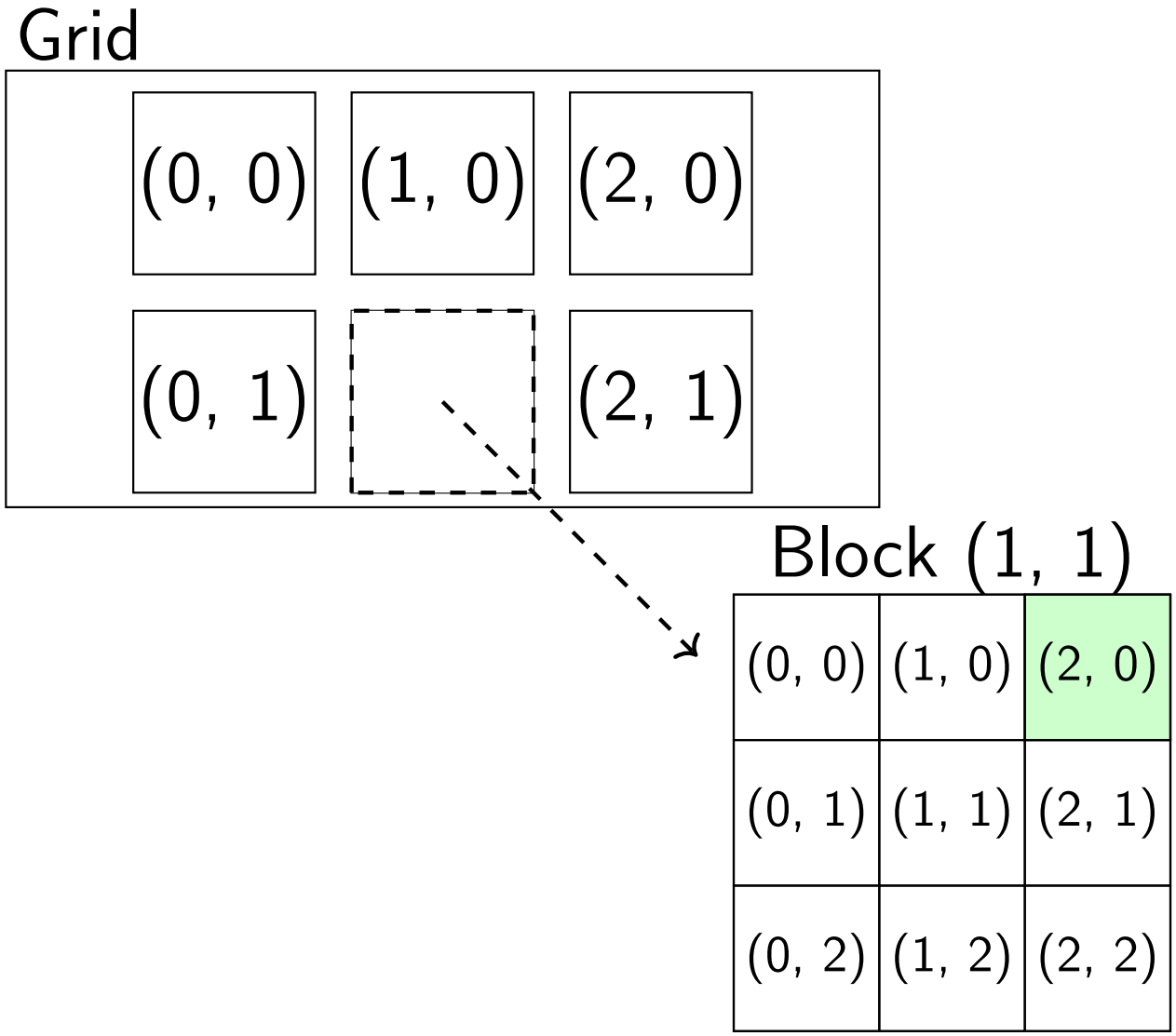
GPU Programming Model XI

- It takes block ID and block size into account.
- The following figure shows the coordinate (x, y) of blocks in a 2×3 grid and threads in a 3×3 block. Note that x is horizontal and y is vertical.
- For example, if we have `blockIdx.x` $= 1$, `threadIdx.x` $= 2$, `blockIdx.y` $= 1$ and `threadIdx.y` $= 0$, we perform `MatAdd` on

  ```
  C[5][3]=A[5][3]+B[5][3]
  ```

  using the Thread (2, 0) in the Block (1, 1) in the Grid, which is the green thread in the following figure.

GPU Programming Model XII



Matrix-matrix Multiplications I

- An optimized implementation must
  - parallelize the operations, and
  - reduce the number of memory accesses.
- It is complicated to achieve both.
- As in earlier slides we have demonstrated the importance of memory accesses, here we focus on the massive parallelization of GPU.
- We borrow many materials from the blog "2678x Faster Matrix Multiplication with a GPU" and its code repository.[2]

Matrix-matrix Multiplications II

- We calculate

$$C = A \times B$$

by

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}.$$

- We let each thread handle one C_{ij} calculation.
- The kernel function is as follows.

Matrix-matrix Multiplications III

```
__global__ void matmul_kernel(float* A,
        float* B, float* C, int N)
{
    int i = blockDim.y*blockIdx.y +
            threadIdx.y;
    int j = blockDim.x*blockIdx.x +
            threadIdx.x;

    if (i < N && j < N)
    {
        float value = 0;
```

Matrix-matrix Multiplications IV

```
            for (int k = 0; k < N; k++)
            {
                value += A[i*N+k] * B[k*N+j];
            }

            C[i*N+j] = value;
        }
    }
```

- We have the following function to call the kernel function.

Matrix-matrix Multiplications V

```
void matmul_gpu(float* A, float* B,
                float* C, int N)
{
    float* d_A; float* d_B; float* d_C;

    cudaMalloc((void**) &d_A,
               N*N*sizeof(float));
    cudaMalloc((void**) &d_B,
               N*N*sizeof(float));
    cudaMalloc((void**) &d_C,
               N*N*sizeof(float));
```

Matrix-matrix Multiplications VI

```
cudaMemcpy(d_A, A, N*N*sizeof(float),
                cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, N*N*sizeof(float),
                cudaMemcpyHostToDevice);

int block_size = 32;
dim3 dim_block(block_size, block_size);
dim3 dim_grid(N/block_size,
                    N/block_size);
matmul_kernel<<<dim_grid, dim_block>>>
```

Matrix-matrix Multiplications VII

```
                    (d_A, d_B, d_C, N);

    cudaMemcpy(C, d_C, N*N*sizeof(float),
                    cudaMemcpyDeviceToHost);

    cudaFree(d_A); cudaFree(d_B);
    cudaFree(d_C);
}
```

- The input includes three matrices `A`, `B`, `C` in CPU.
- Thus, we allocate three matrices in GPU by `cudaMalloc`.

Matrix-matrix Multiplications VIII

- We then copy matrices from CPU to GPU by `cudaMemcpy`.
- Finally, the main program is as follows.

```
#include <stdio.h>
#include <sys/time.h>

int main(int argc, char const *argv[])
{
    int N = 32768;

    struct timeval t1, t2;
```

Matrix-matrix Multiplications IX

```
double elapsedTime;

float* A = (float*)malloc(
           N*N*sizeof(float));
float* B = (float*)malloc(
           N*N*sizeof(float));
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
    {
        A[i*N+j] = (float)(rand()%10);
        B[i*N+j] = (float)(rand()%10);
```

Matrix-matrix Multiplications X

```
    }

float* C = (float*)malloc(
            N*N*sizeof(float));
gettimeofday(&t1, NULL);
matmul_gpu(A, B, C, N);
gettimeofday(&t2, NULL);
elapsedTime = (t2.tv_sec - t1.tv_sec) +
 (t2.tv_usec - t1.tv_usec) / 1000000.0;
printf("GPU time (N: %d): %f sec\n",
       N, elapsedTime);
```

Matrix-matrix Multiplications XI

```
    free(A); free(B); free(C);
    return 0;
}
```

- In the code, we randomly generate values of A and B.
- Once you include all the above programs (main and subroutines) to a fine, we can build an executable file by

```
$ nvcc matmul.cu -o matmul
```

Matrix-matrix Multiplications XII

- We consider a machine with NVIDIA GeForce RTX 3090.
- Running the GPU code gives

```
GPU time (N: 32768): 33.333848 sec
```

- On a high-end CPU (AMD Ryzen 9 7950X 16-Core Processor), using Octave gives

```
> A = B = randn(32768,32768);
> tic; C = A*B; toc
Elapsed time is 74.1317 seconds.
```

- Note that Octave calls highly sophisticated optimized BLAS.

Matrix-matrix Multiplications XIII

- But on GPU, by a very simple implementation, the running time is already shorter.
- If we run optimized BLAS for GPU (e.g., cuBLAS on Nvidia GPUs), the running time should be even less.
- Note that our GPU code achieves reasonably good memory efficiency.
- For each block, we access 32 rows and 32 columns of A and B, respectively, to calculate a 32×32 sub-matrix in C.
- If we reduce the size of each block,

Matrix-matrix Multiplications XIV

```
int block_size = 4;
dim3 dim_block(block_size, block_size);
dim3 dim_grid(N/block_size,
              N/block_size);
```

we see the running time increases:

```
GPU time (N: 32768): 112.512197 sec
```

- The reason is due to worse memory efficiency.

[2] https://0mean1sigma.com/2678x-faster-how-gpus-supercharge-matrix-multiplication/

References I

K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 133–137, 2004.