SSE3054 Multicore Systems

CUDA Memory Model 1

Prof. Seokin Hong

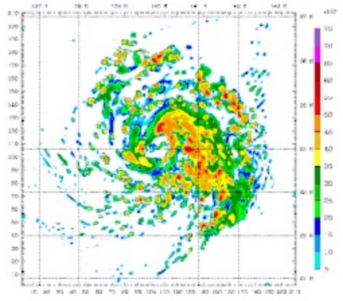
Agenda

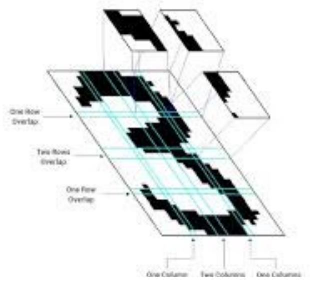
- **Matrix Multiplication**
  - **Basic Version**
  - **Tiled Version**
- Review: Memory Hierarchy
- Importance of Memory Access Efficiency
- GPU Memory Hierarchy
- Improving Tiled Matrix Multiplication
- Impact of Memory on Parallelism

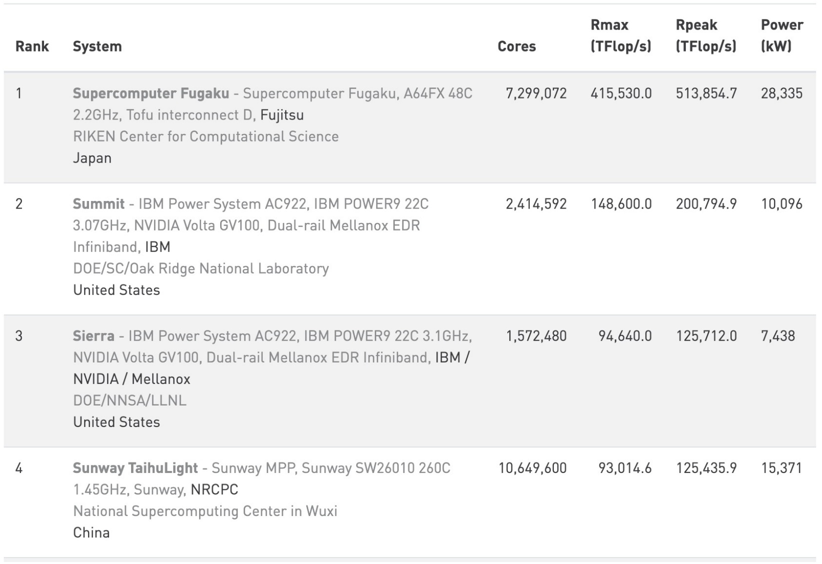
Matrix Multiplication is Fundamental in HPC

- Many algorithmic problems can be solved by means of matrix computation
  - Scientific computing
  - Pattern Recognition
  - Graph Analysis

- LINPACK relies heavily on the matrix multiplication
  - LINPACK is used for measuring the performance of Supercomputer







Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,299,072	415,530.0	513,854.7	28,335
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371

Basic Matrix Multiplication

Matrix Multiplication

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \cdot \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} & b_{04} \\ b_{10} & b_{11} & b_{12} & b_{13} & b_{14} \\ b_{20} & b_{21} & b_{22} & b_{23} & b_{24} \\ b_{30} & b_{31} & b_{32} & b_{33} & b_{34} \\ b_{40} & b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

Matrix Multiplication

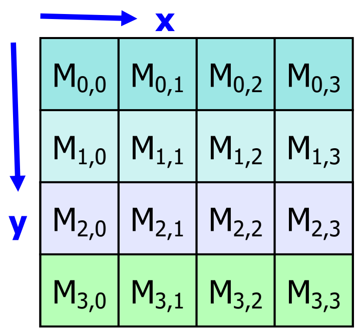
- C_{ij} = dot product of $A_{i\_}$ and $B_{\_j}$

$$\mathbf{C} = \begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} & c_{04} \\ c_{10} & c_{11} & c_{12} & c_{13} & c_{14} \\ c_{20} & c_{21} & c_{22} & c_{23} & c_{24} \\ c_{30} & \boxed{c_{31}} & c_{32} & c_{33} & c_{34} \\ c_{40} & c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \\ \hline a_{40} & a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \cdot \left[\begin{array}{c|c|ccc} b_{00} & b_{01} & b_{02} & b_{03} & b_{04} \\ b_{10} & b_{11} & b_{12} & b_{13} & b_{14} \\ b_{20} & b_{21} & b_{22} & b_{23} & b_{24} \\ b_{30} & b_{31} & b_{32} & b_{33} & b_{34} \\ b_{40} & b_{41} & b_{42} & b_{43} & b_{44} \end{array}\right]$$

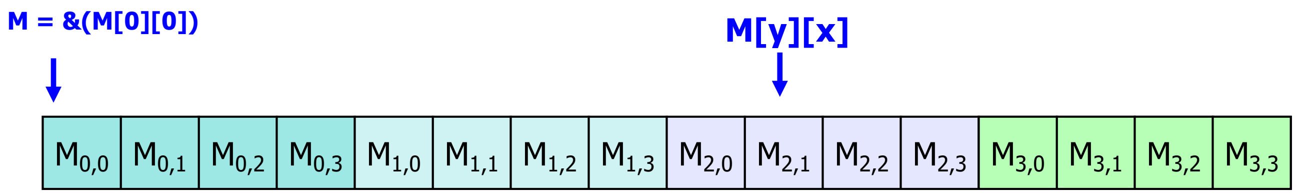
$$c_{31} = \begin{bmatrix} a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix} \cdot \begin{bmatrix} b_{01} \\ b_{11} \\ b_{21} \\ b_{31} \\ b_{41} \end{bmatrix}$$

Row-major Matrix Layout in C/C++

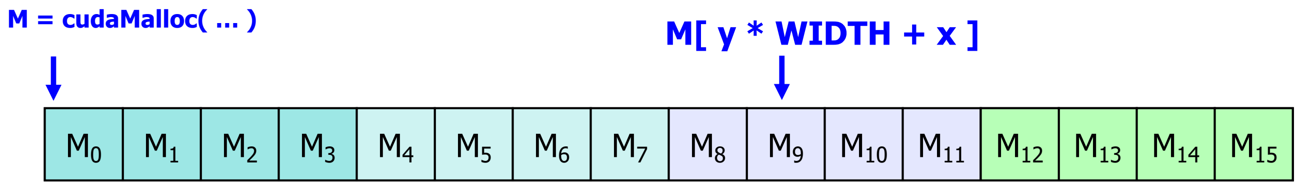
- logical layout:


x
y
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

- physical layout: 1D array


M = &(M[0][0])
M[y][x]
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

- re-interpret:


M = cudaMalloc(…)
M[y * WIDTH + x]
M_0 M_1 M_2 M_3 M_4 M_5 M_6 M_7 M_8 M_9 M_{10} M_{11} M_{12} M_{13} M_{14} M_{15}

Matrix Multiplication

- C_{ij} = dot product of $A_{i\_}$ and $B_{\_j}$

$$c_{31} = \begin{bmatrix} a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix} \cdot \begin{bmatrix} b_{01} \\ b_{11} \\ b_{21} \\ b_{31} \\ b_{41} \end{bmatrix}$$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

```
int sum = 0;

for (int k = 0; k < WIDTH; ++k) {
            sum += a[i][k] * b[k][j];
}

c[i][j] = sum;
```

matmul-host.cu (1/2)

- calculate matrix multiplication on CPU

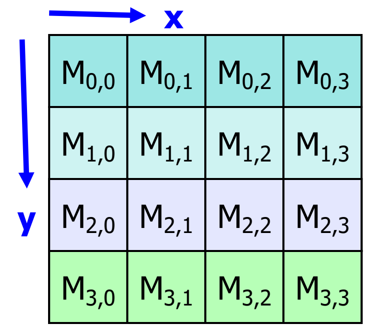
```
//prepare data
 const int WIDTH = 5;
 int a[WIDTH][WIDTH];
 int b[WIDTH][WIDTH];
 int c[WIDTH][WIDTH] = { 0 };

//make matrices A, B
for (int y = 0; y < WIDTH; ++y) {
                    for (int x = 0; x < WIDTH; ++x) {
                              a[y][x] = y + x;
                              b[y][x] = y + x;
                    }
          }
```

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \end{bmatrix}$$

matmul-host.cu (2/2)

- calculate matrix multiplication on CPU

```
//calculation code
for (int y = 0; y < WIDTH; ++y) {
        for (int x = 0; x < WIDTH; ++x) {
                int sum = 0;
                for (int k = 0; k < WIDTH; ++k) {
                        sum += a[y][k] * b[k][x];
                }
                c[y][x] = sum;
        }
}
```


x
y
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \end{bmatrix}$$

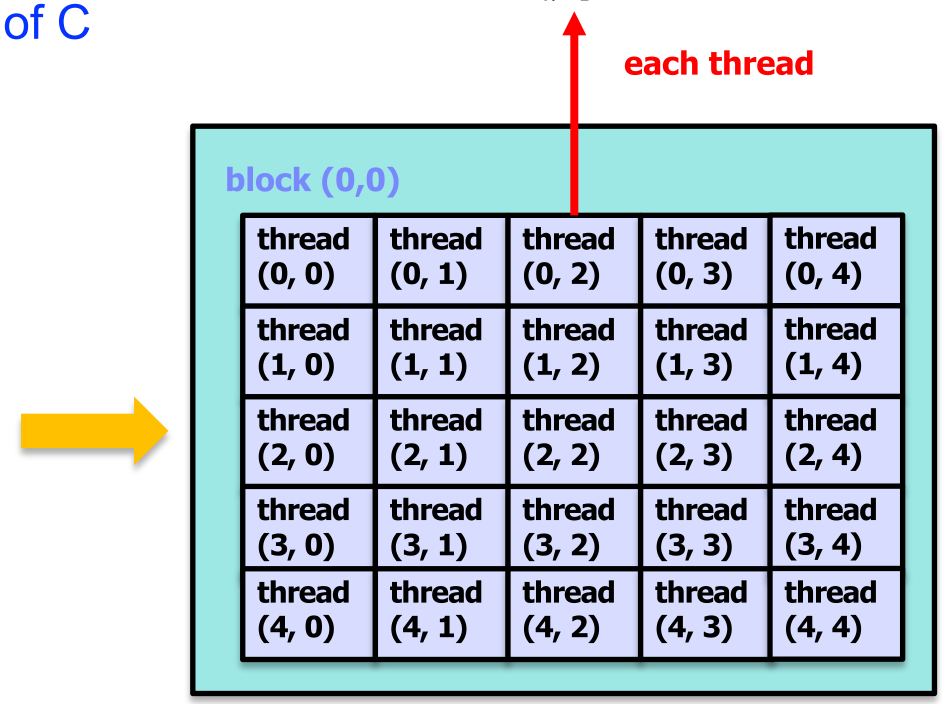
CUDA Matrix Multiplication

- C = A * B
  - Size: WIDTH x WIDTH

- Thread organization (layout)
  - One thread handles one element of C

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

$$\mathbf{C} = \begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} & c_{04} \\ c_{10} & c_{11} & c_{12} & c_{13} & c_{14} \\ c_{20} & c_{21} & c_{22} & c_{23} & c_{24} \\ c_{30} & c_{31} & c_{32} & c_{33} & c_{34} \\ c_{40} & c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$



CUDA Matrix Multiplication (1/2)

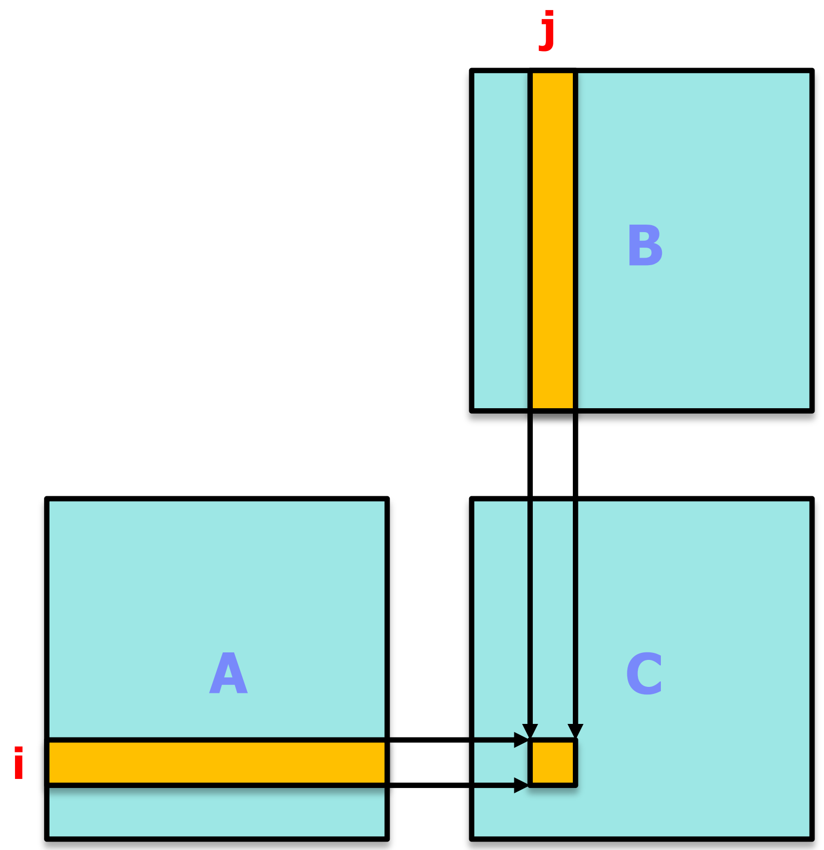
- C = A * B
  - Size: WIDTH x WIDTH

$$\mathbf{C} = \begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} & c_{04} \\ c_{10} & c_{11} & c_{12} & c_{13} & c_{14} \\ c_{20} & c_{21} & c_{22} & c_{23} & c_{24} \\ c_{30} & \boxed{c_{31}} & c_{32} & c_{33} & c_{34} \\ c_{40} & c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ \boxed{a_{30}} & \boxed{a_{31}} & \boxed{a_{32}} & \boxed{a_{33}} & \boxed{a_{34}} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \cdot \begin{bmatrix} b_{00} & \boxed{b_{01}} & b_{02} & b_{03} & b_{04} \\ b_{10} & \boxed{b_{11}} & b_{12} & b_{13} & b_{14} \\ b_{20} & \boxed{b_{21}} & b_{22} & b_{23} & b_{24} \\ b_{30} & \boxed{b_{31}} & b_{32} & b_{33} & b_{34} \\ b_{40} & \boxed{b_{41}} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

- **Memory usage view:**
  - read from A and B
  - to calculate:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

  - using:
    - **a[i * WIDTH + k]**
    - **b[k * WIDTH + j];**

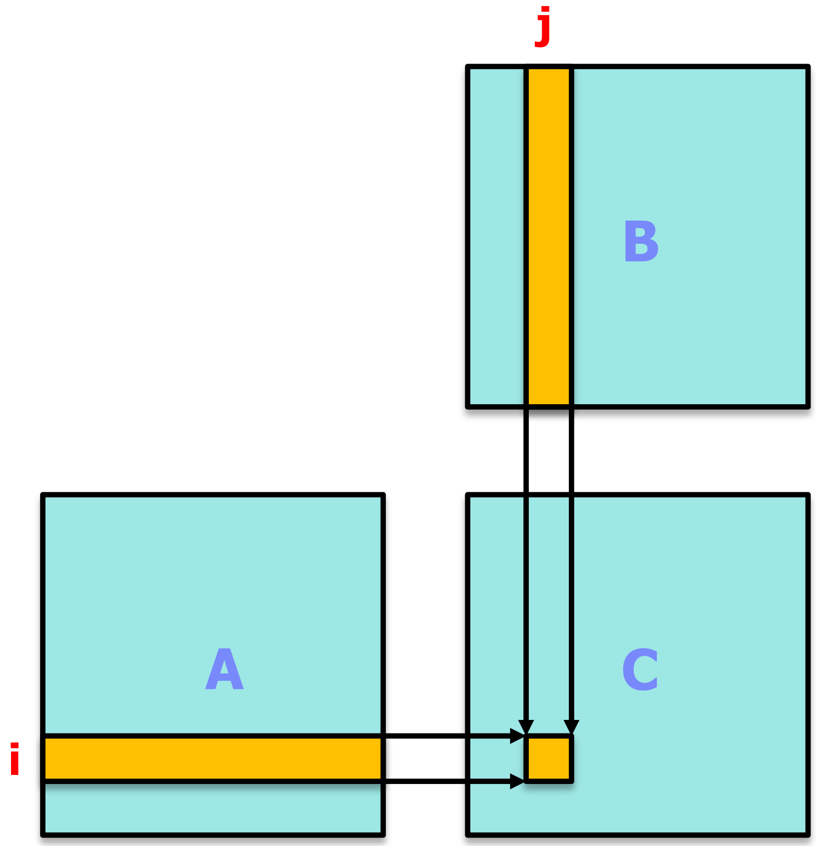


CUDA Matrix Multiplication (2/2)

- **One Block of threads compute matrix C**
  - Each thread computes one element of C

- **Each thread**
  - loads a row of matrix A
  - loads a column of matrix B
  - Perform one multiply and addition for each pair of A and B elements

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$



matmul-dev.cu (1/3) : Kernel code

- **use WIDTH * WIDTH threads**

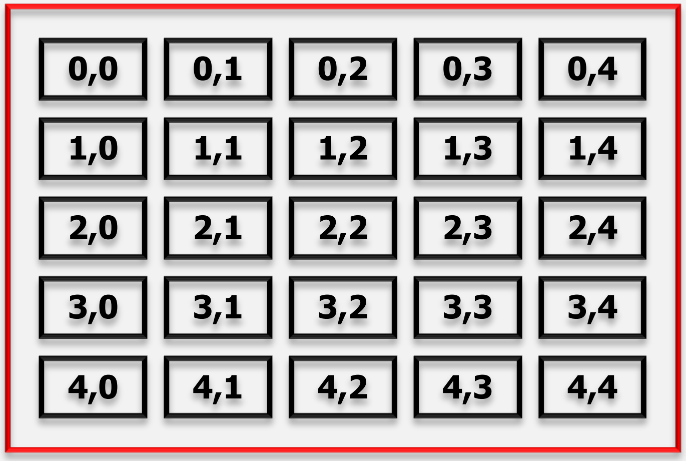
```
__global__ void mulKernel(int* c, const int* a, const int* b, const int WIDTH) {
        int x = threadIdx.x;
        int y = threadIdx.y;

        int i = y * WIDTH + x;                          // [y][x] = y * WIDTH + x;

        int sum = 0;
        for (int k = 0; k < WIDTH; ++k) {
                sum += a[y * WIDTH + k] * b[k * WIDTH + x];
        }
        c[i] = sum;
}
```

matmul-dev.cu (2/3) : Host Code

```
int main(void) {
    // host-side data
    const int WIDTH = 5;
    int a[WIDTH][WIDTH];
    int b[WIDTH][WIDTH];
    int c[WIDTH][WIDTH] = { 0 };
    // make a, b matrices
    for (int y = 0; y < WIDTH; ++y) {
                for (int x = 0; x < WIDTH; ++x) {
                            a[y][x] = y * 10 + x;
                            b[y][x] = (y * 10 + x) * 100;
                }
    }

    // device-side data
    int *dev_a = 0;
    int *dev_b = 0;
    int *dev_c = 0;
    //allocate device memory
    cudaMalloc((void**)&dev_a, WIDTH * WIDTH * sizeof(int));
    cudaMalloc((void**)&dev_b, WIDTH * WIDTH * sizeof(int));
    cudaMalloc((void**)&dev_c, WIDTH * WIDTH * sizeof(int));

    // copy from host to device
    cudaMemcpy(dev_a, a, WIDTH * WIDTH * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, WIDTH * WIDTH * sizeof(int), cudaMemcpyHostToDevice);
```

matmul-dev.cu (3/3) : Host Code

```
    // launch a kernel on the GPU
    dim3   dimBlock(WIDTH, WIDTH, 1); // x, y, z
    mulKernel<<<1, dimBlock>>>(dev_c, dev_a, dev_b, WIDTH);
    CUDA_CHECK( cudaPeekAtLastError() );

    // copy from device to host
    cudaMemcpy(c, dev_c, WIDTH * WIDTH * sizeof(int),
                        cudaMemcpyDeviceToHost);
    // free device memory
    cudaFree(dev_c);
    cudaFree(dev_a);
    cudaFree(dev_b);

    // print the result
    for (int y = 0; y < WIDTH; ++y) {
        for (int x = 0; x < WIDTH; ++x) {
            printf("%5d", c[y][x]);
        }
        printf("\n");
    }
    return 0;
}
```



Tiled Matrix Multiplication

Slide Credit: Slides are modified from Prof Baek's slides

Thread Organization in the Simple Matrix Multiplication

- matrix → 2D layout
- small size matrix → a single block !

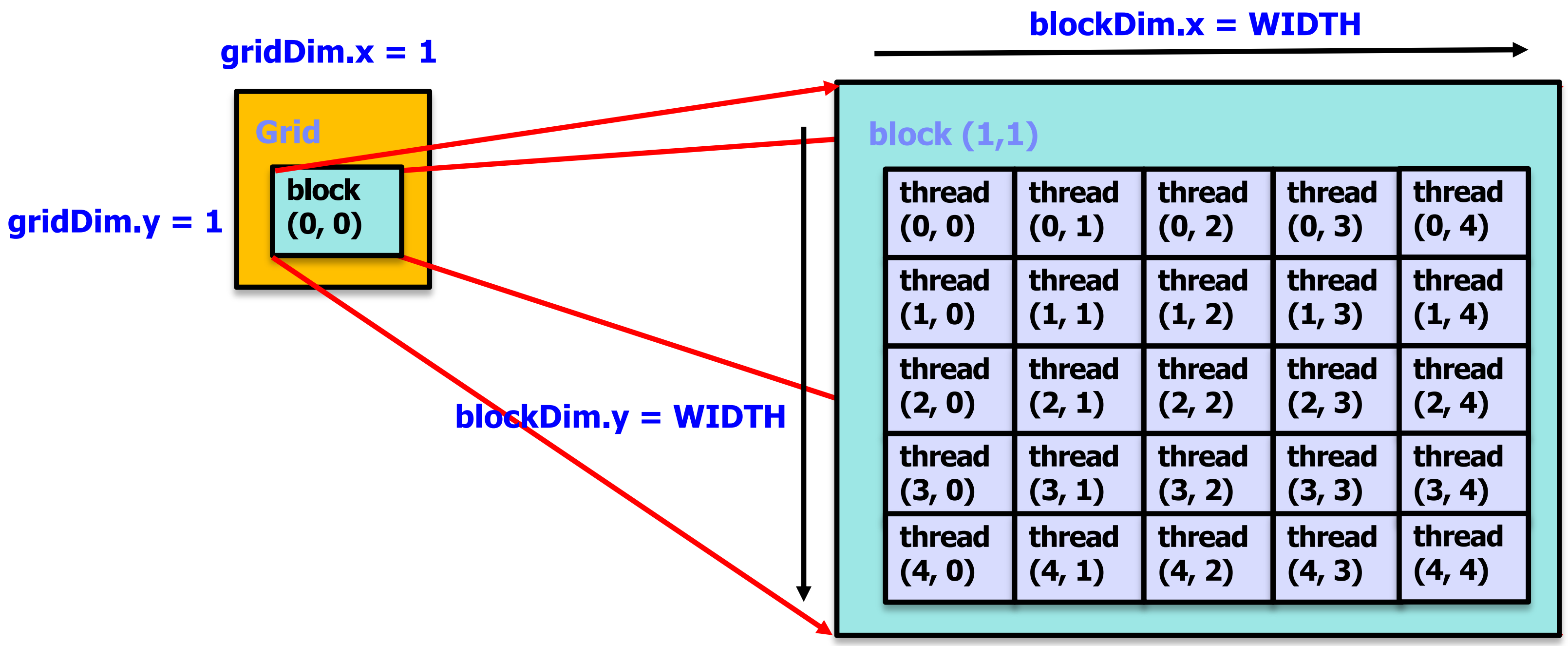


Image from http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

Any Problem?

- We used only one thread block...
- Each thread block can execute at most 1024 threads
  - some old architecture can execute only 256 or 512 threads
- So, maximum matrix size is...
  - 32 x 32 = 1024
  - with a single thread block...
- Many global memory accesses
  - Global memory is slow
  - We will discuss it later!
- solution?
  - use multiple thread blocks !

Thread Organization (layout)

- matrix → 2D layout
- tiled approach : use multiple blocks

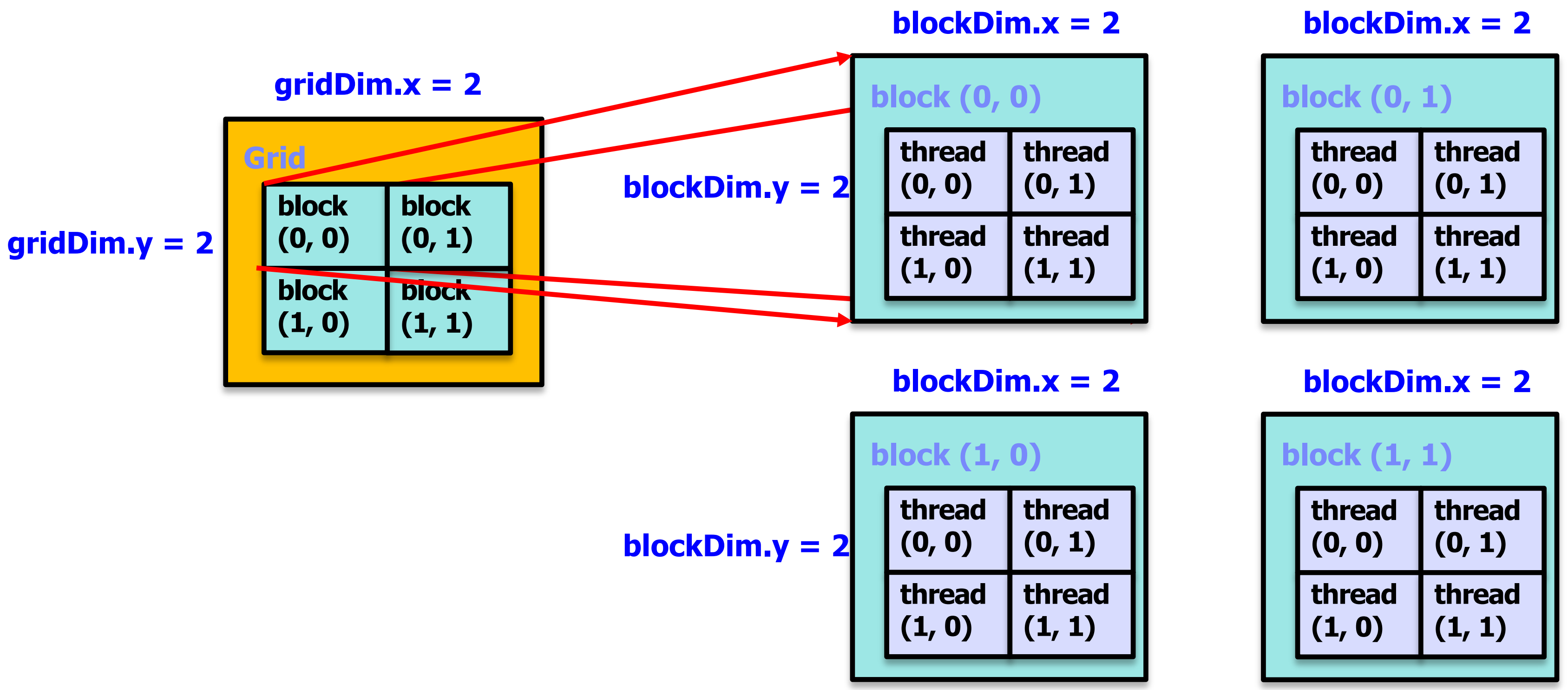


Image from http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

Simplified Thread Organization

- assumption: square matrices
- global size : WIDTH x WIDTH
- tile → a block : TILE_WIDTH x TILE_WIDTH
- grid : ceil(WIDTH / TILE_WIDTH) x ceil(WIDTH / TILE_WIDTH)

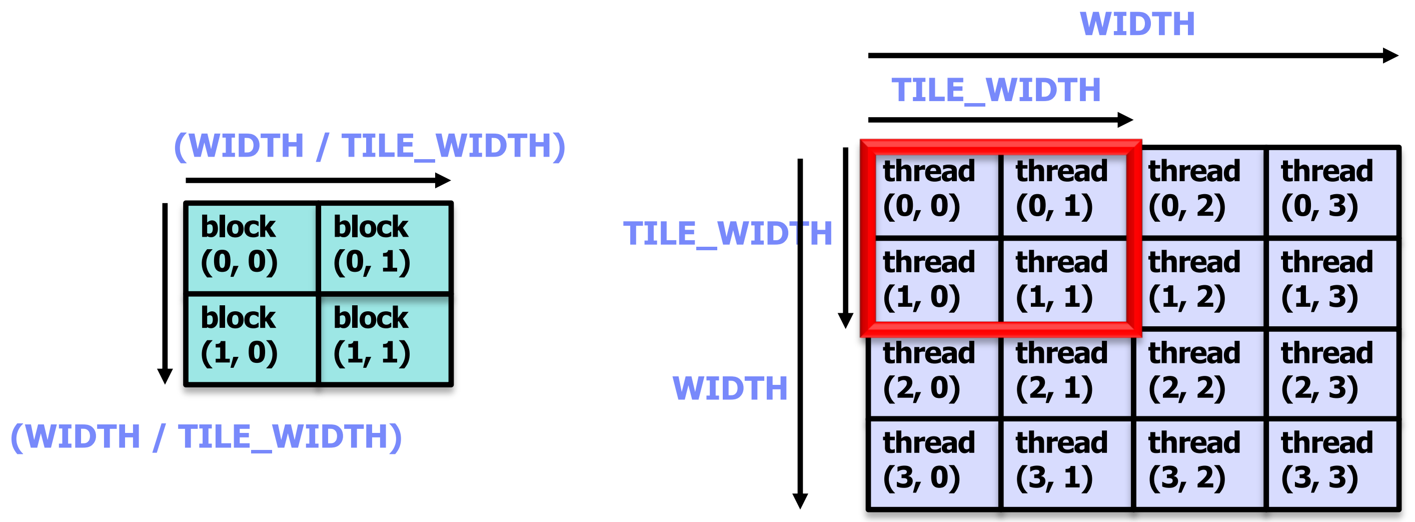
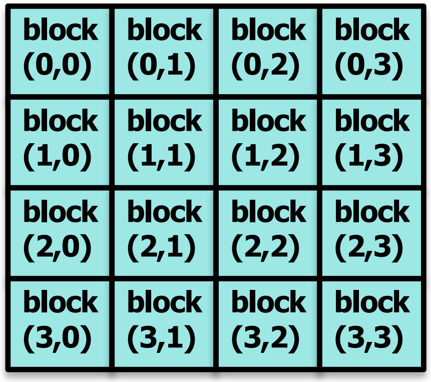
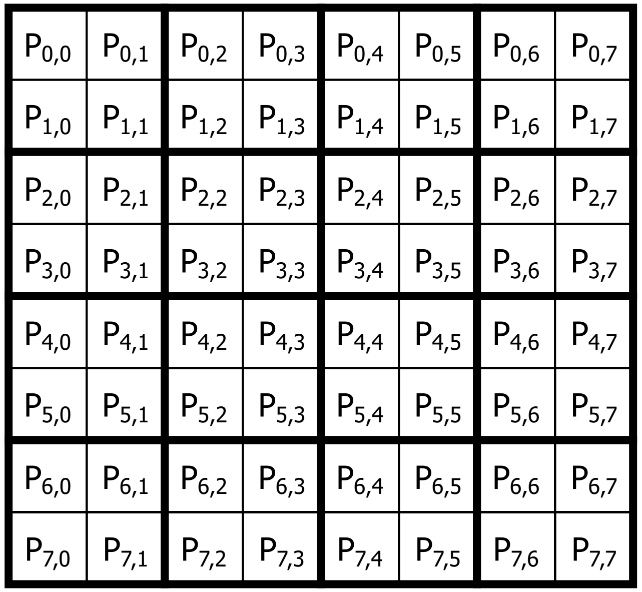


Image from http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

An Example of Thread Organization

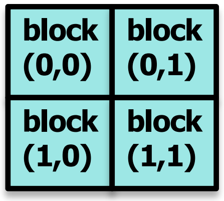
- WIDTH = 8
- TILE_WIDTH = 2
- grid dimension = 4 x 4

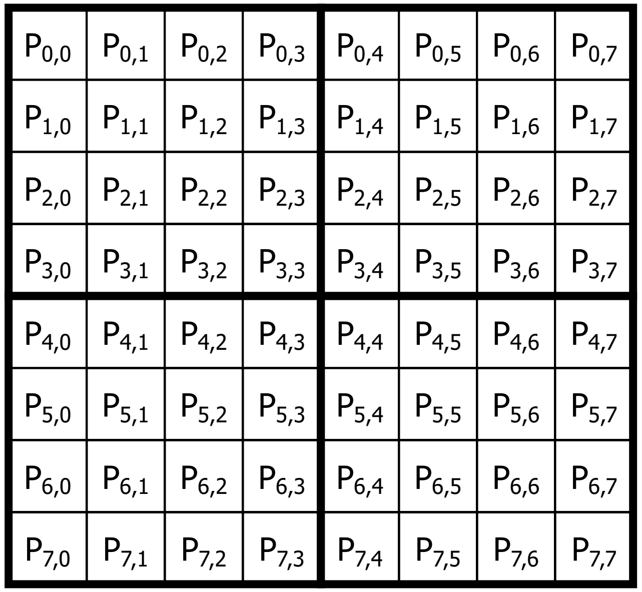



$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$ $P_{0,4}$ $P_{0,5}$ $P_{0,6}$ $P_{0,7}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$ $P_{1,4}$ $P_{1,5}$ $P_{1,6}$ $P_{1,7}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$ $P_{2,4}$ $P_{2,5}$ $P_{2,6}$ $P_{2,7}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$ $P_{3,4}$ $P_{3,5}$ $P_{3,6}$ $P_{3,7}$
$P_{4,0}$ $P_{4,1}$ $P_{4,2}$ $P_{4,3}$ $P_{4,4}$ $P_{4,5}$ $P_{4,6}$ $P_{4,7}$
$P_{5,0}$ $P_{5,1}$ $P_{5,2}$ $P_{5,3}$ $P_{5,4}$ $P_{5,5}$ $P_{5,6}$ $P_{5,7}$
$P_{6,0}$ $P_{6,1}$ $P_{6,2}$ $P_{6,3}$ $P_{6,4}$ $P_{6,5}$ $P_{6,6}$ $P_{6,7}$
$P_{7,0}$ $P_{7,1}$ $P_{7,2}$ $P_{7,3}$ $P_{7,4}$ $P_{7,5}$ $P_{7,6}$ $P_{7,7}$

An Example of Thread Organization

- WIDTH = 8
- TILE_WIDTH = 4
- grid dimension = 2 x 2





$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$	$P_{0,4}$	$P_{0,5}$	$P_{0,6}$	$P_{0,7}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$	$P_{1,6}$	$P_{1,7}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$	$P_{2,6}$	$P_{2,7}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$	$P_{3,6}$	$P_{3,7}$
$P_{4,0}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$	$P_{4,6}$	$P_{4,7}$
$P_{5,0}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$
$P_{6,0}$	$P_{6,1}$	$P_{6,2}$	$P_{6,3}$	$P_{6,4}$	$P_{6,5}$	$P_{6,6}$	$P_{6,7}$
$P_{7,0}$	$P_{7,1}$	$P_{7,2}$	$P_{7,3}$	$P_{7,4}$	$P_{7,5}$	$P_{7,6}$	$P_{7,7}$

Kernel Launch

```
const int   WIDTH = 8;
const int   TILE_WIDTH = 4;


// Setup the execution configuration
dim3  dimGrid( ceil(WIDTH / TILE_WIDTH), ceil(WIDTH / TILE_WIDTH), 1 );
dim3  dimBlock( TILE_WIDTH, TILE_WIDTH, 1 );


// Launch the device computation threads!
MatrixMulKernel <<<dimGrid, dimBlock>>> (dev_c, dev_a, dev_b, WIDTH);
```

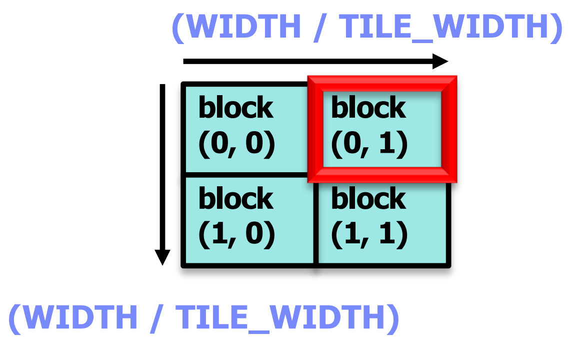
Local Index vs Global Index

- For thread (1,2) of Block (0, 1)
  - blockIdx.y = 0
  - blockIdx.x = 1
- local index
  - threadIdx.y = 1
  - threadIdx.x = 0

- global index
  - y = blockIdx.y * blockDim.y + threadIdx.y

    0 * 2 + 1 → 1
  - x = blockIdx.x * blockDim.x + threadIdx.x

    1 * 2 + 0 → 2



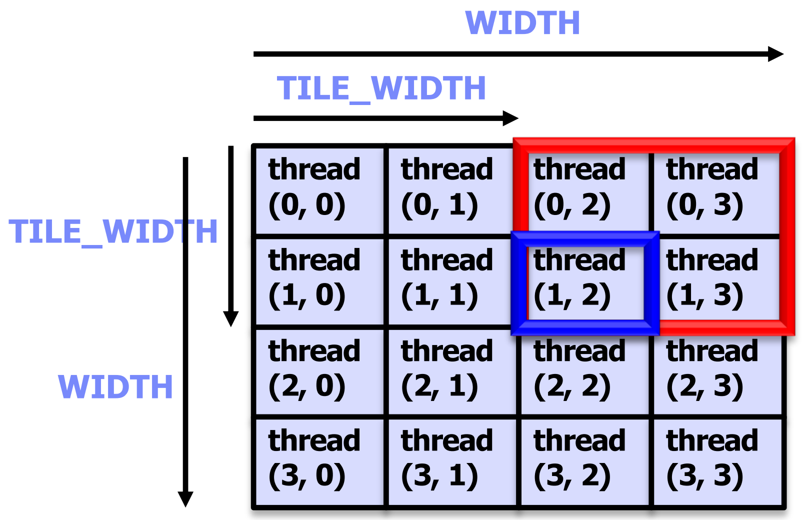
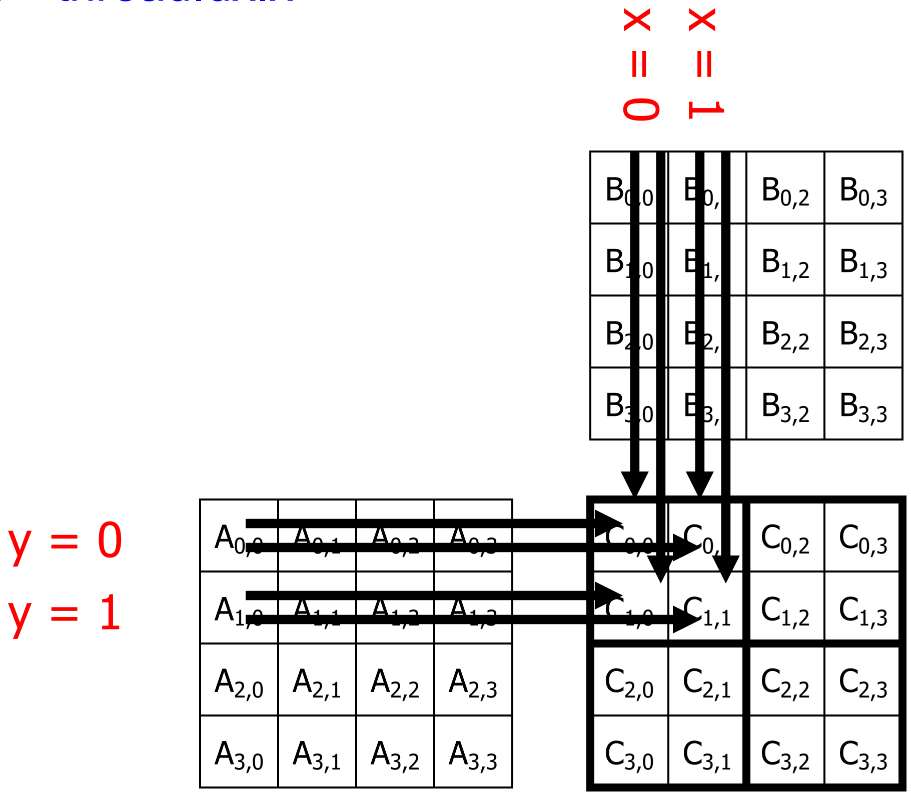


Image from http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

Indices for Block (0,0)

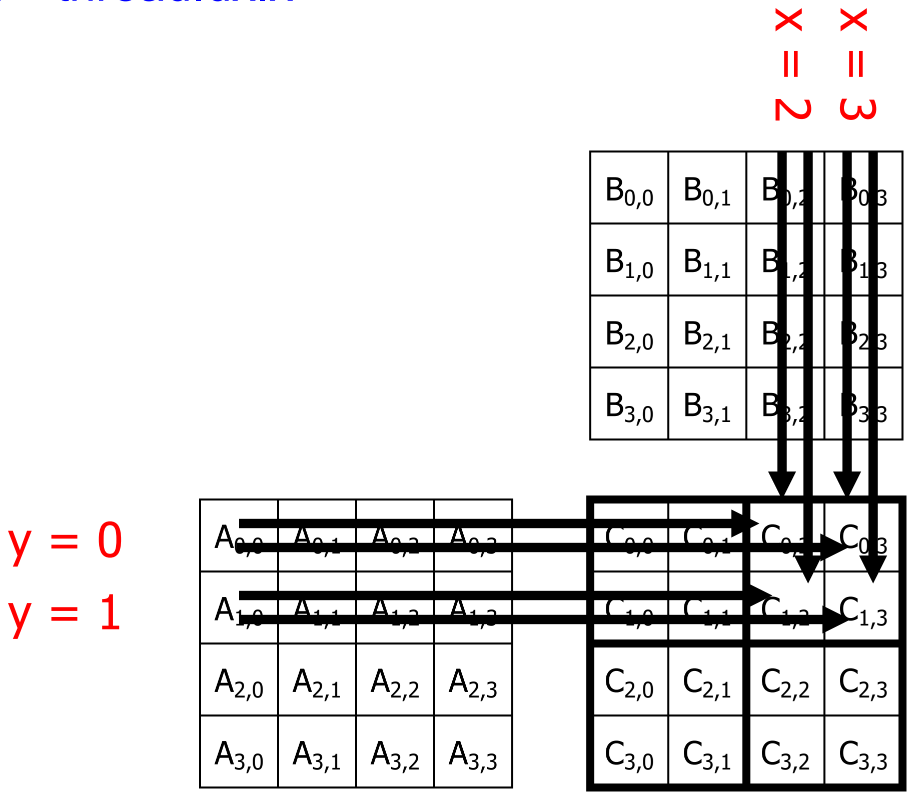
- general calculation:
  - y = blockIdx.y * blockDim.y + threadIdx.y
  - x = blockIdx.x * blockDim.x + threadIdx.x
- in the block (0,0)
  - y = 0 * 2 + threadIdx.y
  - x = 0 * 2 + threadIdx.x



x = 0
x = 1
y = 0
y = 1
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$
$B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$
$B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$
$B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$
$A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$
$A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
$C_{0,0}$ $C_{0,1}$ $C_{0,2}$ $C_{0,3}$
$C_{1,0}$ $C_{1,1}$ $C_{1,2}$ $C_{1,3}$
$C_{2,0}$ $C_{2,1}$ $C_{2,2}$ $C_{2,3}$
$C_{3,0}$ $C_{3,1}$ $C_{3,2}$ $C_{3,3}$

Indices for Block (0,1)

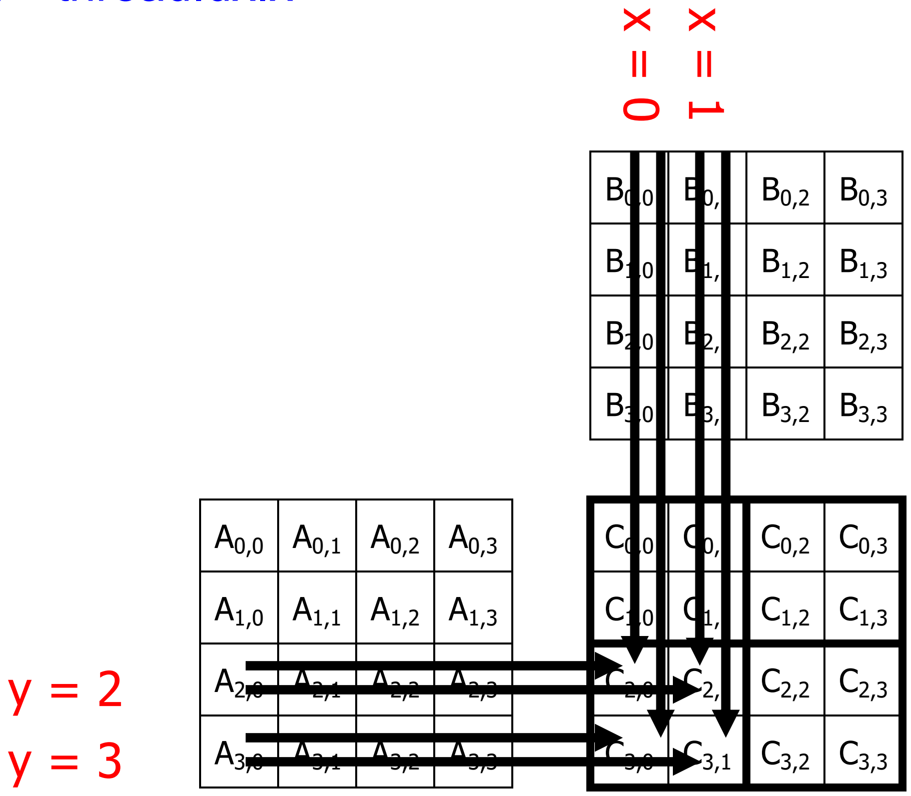
- general calculation:
  - y = blockIdx.y * blockDim.y + threadIdx.y
  - x = blockIdx.x * blockDim.x + threadIdx.x
- in the block (0,1)
  - y = 0 * 2 + threadIdx.y
  - x = 1 * 2 + threadIdx.x



x = 2
x = 3
y = 0
y = 1
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$
$B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$
$B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$
$B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$
$A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$
$A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
$C_{0,0}$ $C_{0,1}$ $C_{0,2}$ $C_{0,3}$
$C_{1,0}$ $C_{1,1}$ $C_{1,2}$ $C_{1,3}$
$C_{2,0}$ $C_{2,1}$ $C_{2,2}$ $C_{2,3}$
$C_{3,0}$ $C_{3,1}$ $C_{3,2}$ $C_{3,3}$

Indices for Block (1,0)

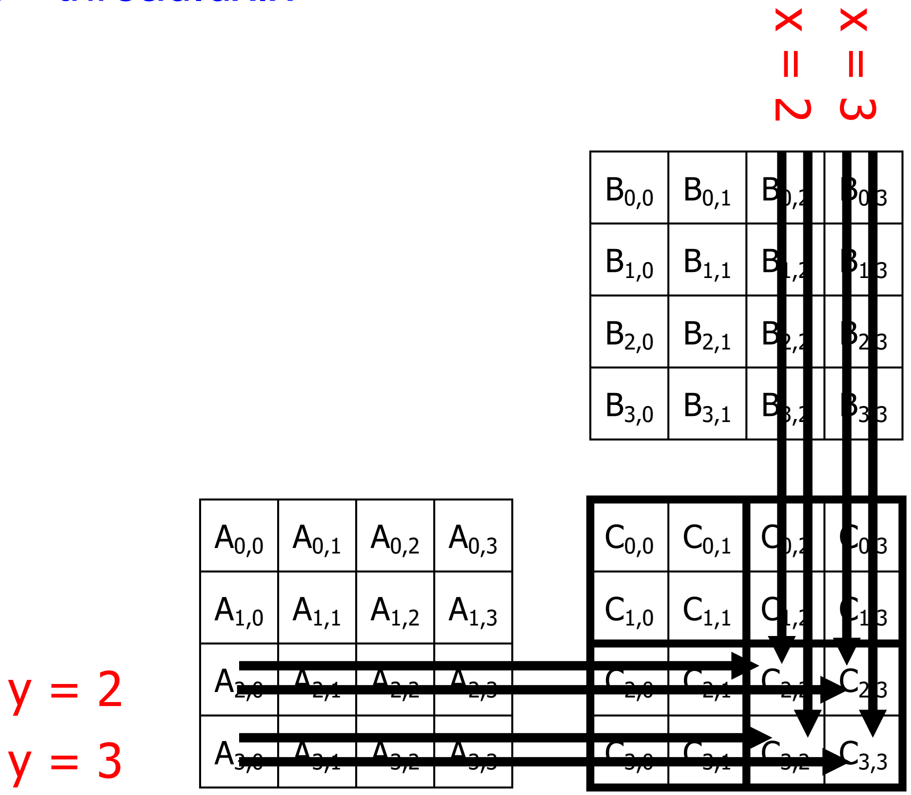
- general calculation:
  - y = blockIdx.y * blockDim.y + threadIdx.y
  - x = blockIdx.x * blockDim.x + threadIdx.x
- in the block (1,0)
  - y = 1 * 2 + threadIdx.y
  - x = 0 * 2 + threadIdx.x



x = 0
x = 1
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$
$B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$
$B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$
$B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$
$A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$
$A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
$C_{0,0}$ $C_{0,1}$ $C_{0,2}$ $C_{0,3}$
$C_{1,0}$ $C_{1,1}$ $C_{1,2}$ $C_{1,3}$
$C_{2,0}$ $C_{2,1}$ $C_{2,2}$ $C_{2,3}$
$C_{3,0}$ $C_{3,1}$ $C_{3,2}$ $C_{3,3}$
y = 2
y = 3

Indices for Block (1,1)

- general calculation:
  - y = blockIdx.y * blockDim.y + threadIdx.y
  - x = blockIdx.x * blockDim.x + threadIdx.x
- in the block (1,1)
  - y = 1 * 2 + threadIdx.y
  - x = 1 * 2 + threadIdx.x



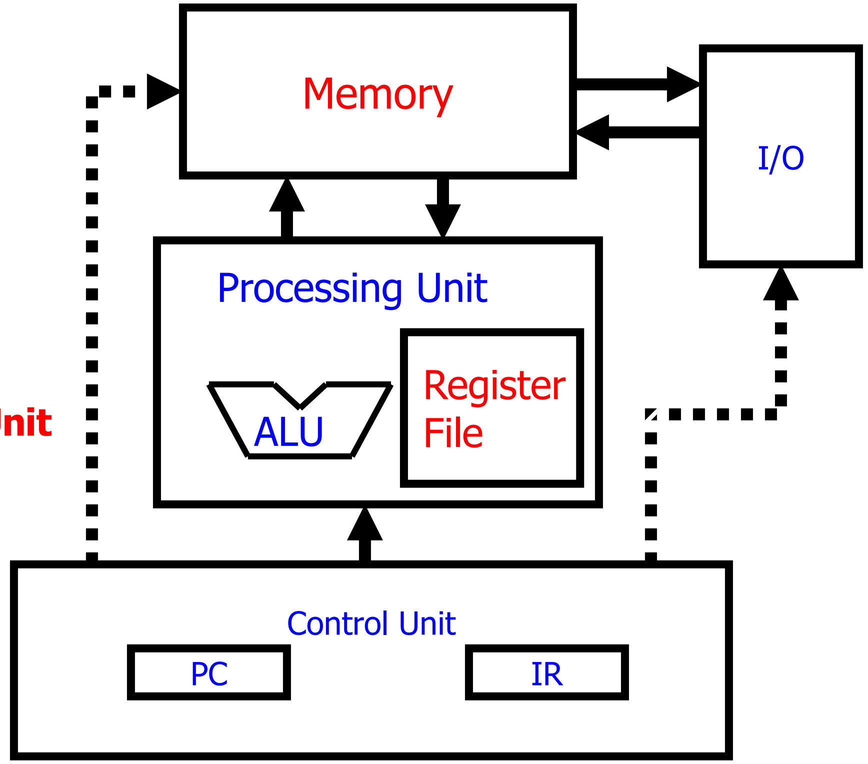
x = 2
x = 3
y = 2
y = 3
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$
$B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$
$B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$
$B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$
$A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$
$A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
$C_{0,0}$ $C_{0,1}$ $C_{0,2}$ $C_{0,3}$
$C_{1,0}$ $C_{1,1}$ $C_{1,2}$ $C_{1,3}$
$C_{2,0}$ $C_{2,1}$ $C_{2,2}$ $C_{2,3}$
$C_{3,0}$ $C_{3,1}$ $C_{3,2}$ $C_{3,3}$

Matrix Multiplication Kernel

```
__global__ void matmul(float* c, const float* a, const float* b, const int width) {
    int  y = blockIdx.y * blockDim.y + threadIdx.y;
    int  x = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0F;
    for (register int k = 0; k < width; ++k) {
        float lhs = a[y * width + k];
        float rhs = b[k * width + x];
        sum += lhs * rhs;
    }
    c[y * width + x] = sum;
}
```

Memory Hierarchy

Why Memory?: Von-Neumann Model



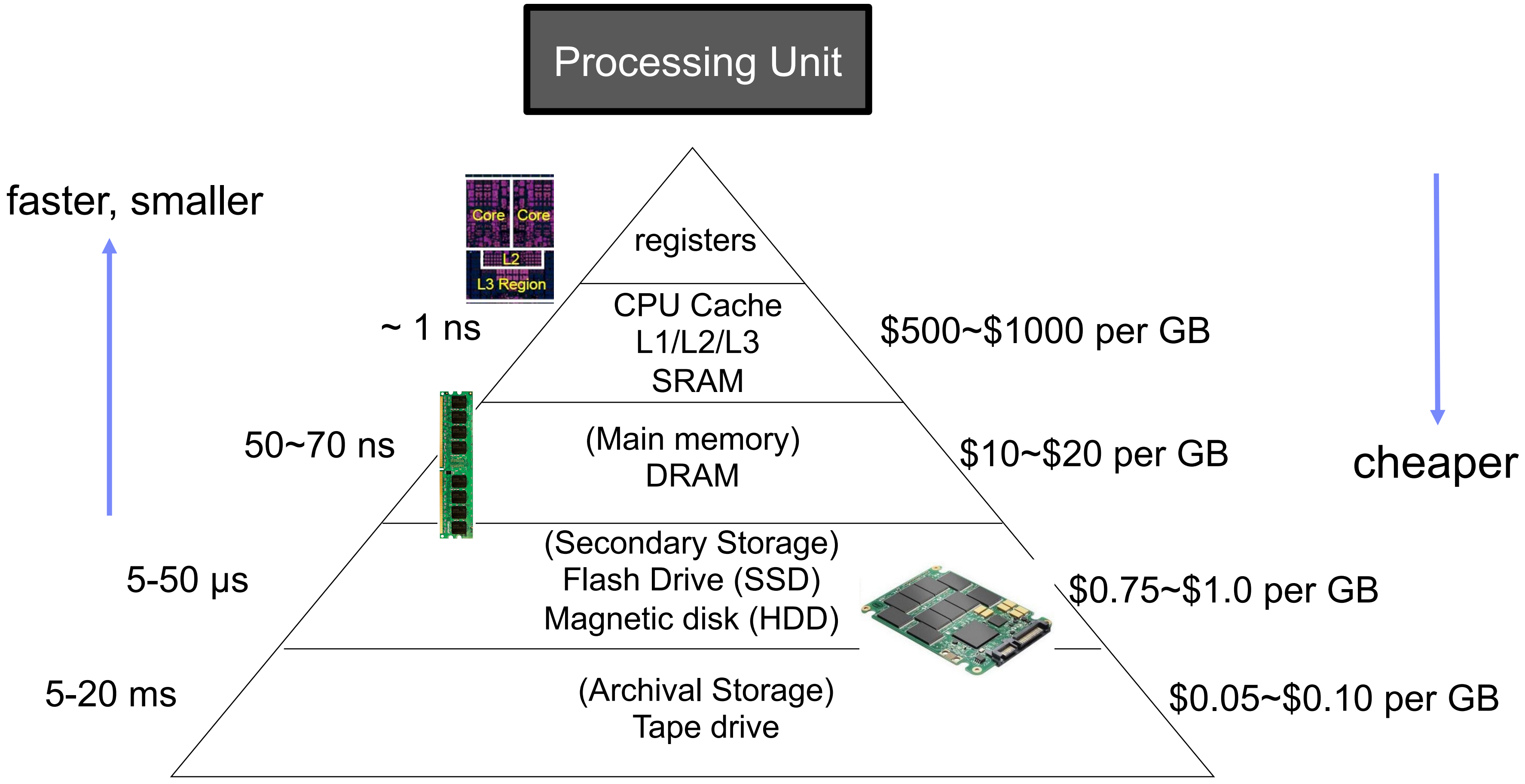
ALU : Arithmetic-Logic Unit

PC: program counter

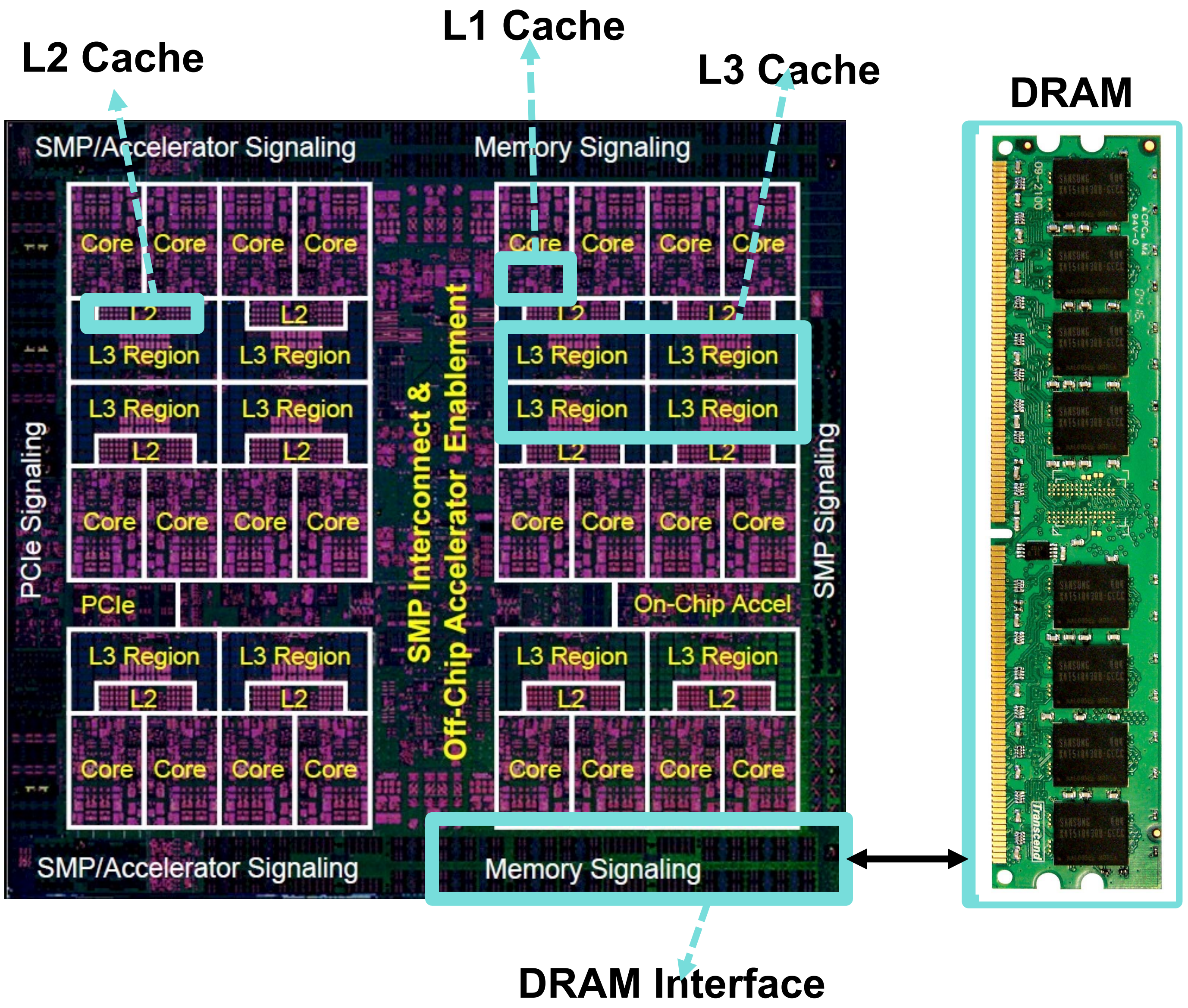
IR: instruction register

Memory Hierarchy

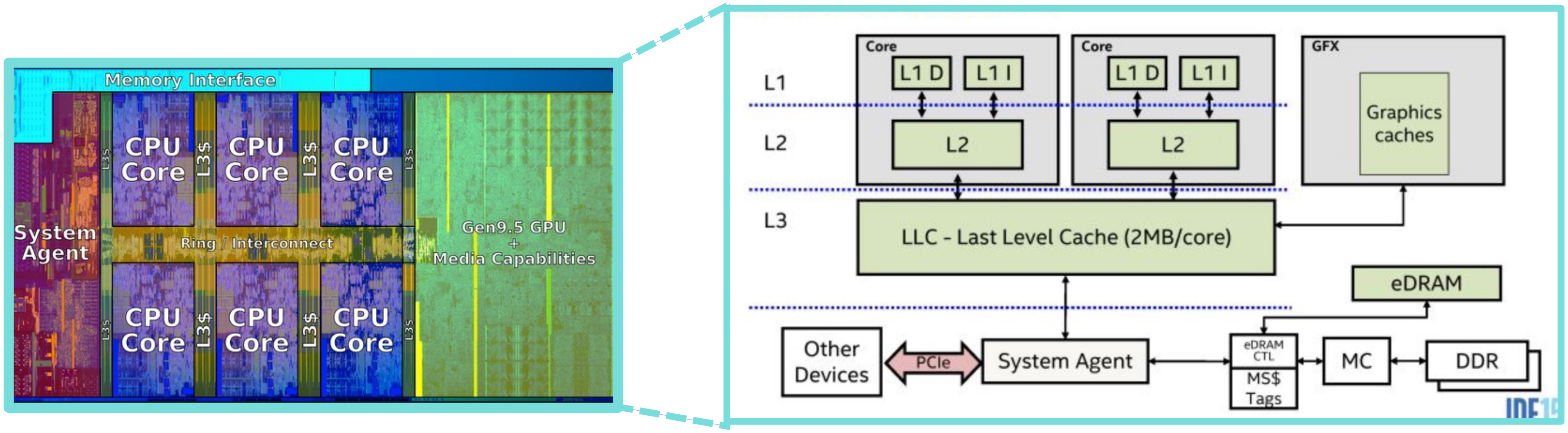
- How to create illusion of the ideal memory (fast, big, cheap)?
  - Have **multiple levels** of memory **with different** speeds and sizes and ensure most of the **data is kept in the fast level**



CPU Memory Hierarchy



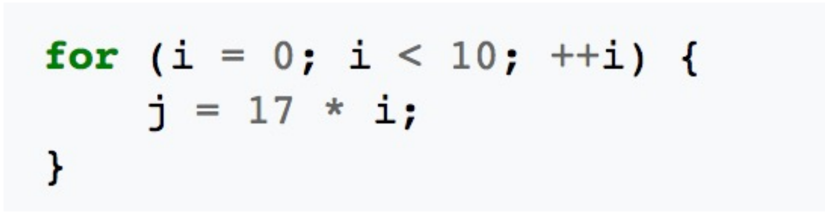
CPU Memory Hierarchy



- **L1I Cache:** 32~64 KB/core
- **L1D Cache**: 32~64KB/core
- **L2 Cache:** 256KB~1MB
- **L3 Cache:** 1~2MB/core
- Some CPUs use L3 Cache

Why does the Memory Hierarchy work well?

- Answer: **Locality**
  - Programs access a relatively small portion of their address space at any instant of time



```
for (i = 0; i < 10; ++i) {
     j = 17 * i;
}
```

- Two different types of locality
  - **Temporal locality**
    - Data accessed recently are likely to be accessed again soon
    - E.g., instructions in a loop, induction variables
  - **Spatial locality**
    - Data near those accessed recently are likely to be accessed soon
    - E.g., sequential instruction access, array data

Importance of Memory Access Efficiency

Poor Memory Access → Poor Performance

- **The poor performance is attributable to**
  - The long access latencies (hundreds of clock cycles)
  - Finite access bandwidth (Giga bytes per second) of global memory

- **Compute-to-global-memory-access-ratio**
  - The number of calculations performed per each access to the global memory
  - used to calculate the expected performance level of a kernel code
  - **High compute-to-global-memory-access-ratio**

    → High performance

Expected Performance of the Matrix Multiplication Kernel

- **Assumption**
  - Matrix A and B are in the global memory (slow but big)
  - Bandwidth of Global memory : 1000GB/s = 250 giga single-precision numbers per second
- **Compute-to-global-memory-access-ratio** of for-loop
  - # of Computation: 1 Addition + 1 Multiplication = 2
  - # of Global memory accesses : 1 for A and 1 for B = 2
  - **Compute-to-global-memory-access-ratio = 1**

```
.....
 for (register int k = 0; k < width; ++k)
 {
         float lhs = a[y * width + k];
         float rhs = b[k * width + x];
         sum += lhs * rhs;
 }
 .....
```

- Expected Performance of for-loop = 250 GFLOP
  - Because compute-to-global-memory-access-ratio is 1 and 250 giga single-precision numbers can be brought from global memory
- If peak performance of GPU is 12TFLOPS,
  - compute-to-global-memory-access-ratio should be 48 or higher

*GFLOP : Giga Floating Point Operations per Second

Agenda

- **Matrix Multiplication**
  - **Simple Version**
  - **Tiled Version**

- Review: Memory Hierarchy
- Importance of Memory Access Efficiency

- GPU Memory Hierarchy
- Improving Tiled Matrix Multiplication
- Impact of Memory on Parallelism