Programming Parallel Processors

Rodrigo Dominguez
Ph.D student
Northeastern University
March 18, 2010

Goals

- Understand General Purpose Computing on GPU (a.k.a. GPGPU)
- Experience CUDA GPU programming
- Understand how massively multi-threaded parallel programming works
- Think about solving a problem in a parallel fashion
- Experience the computational power of GPUs
- Experience the challenges in efficient parallel programming

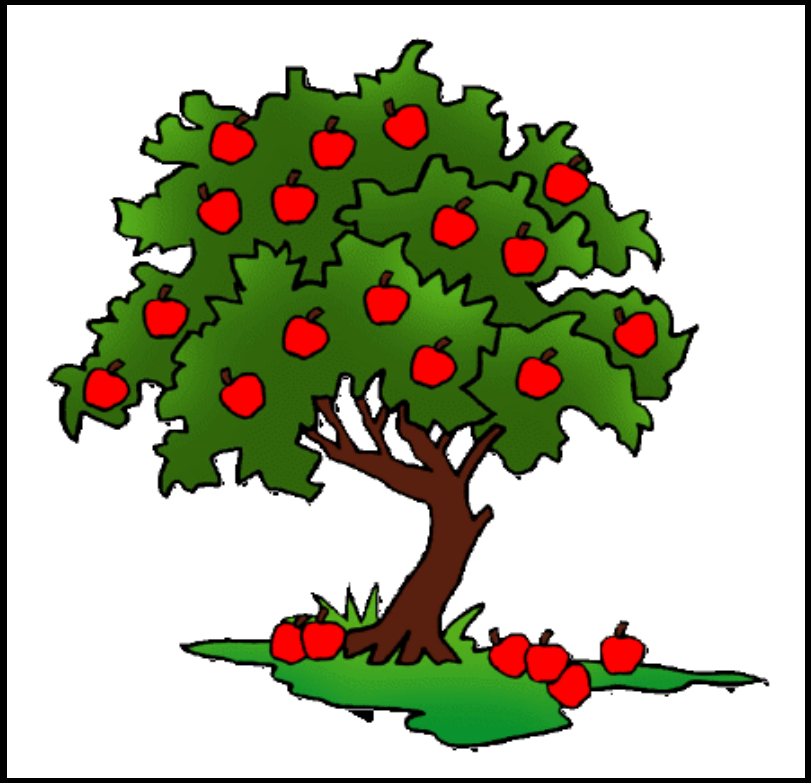
Outline

- GPU: Graphics Processing Unit
- CUDA: Programming Model
- Application 1: Image Rotation
  - Introduction and Design (15 min)
  - Preparation (5 min)
    - Installing a skeleton code, compile test, image view test
  - Hands-on Programming (30 min)
    - Replace ??? with your own CUDA code

Outline

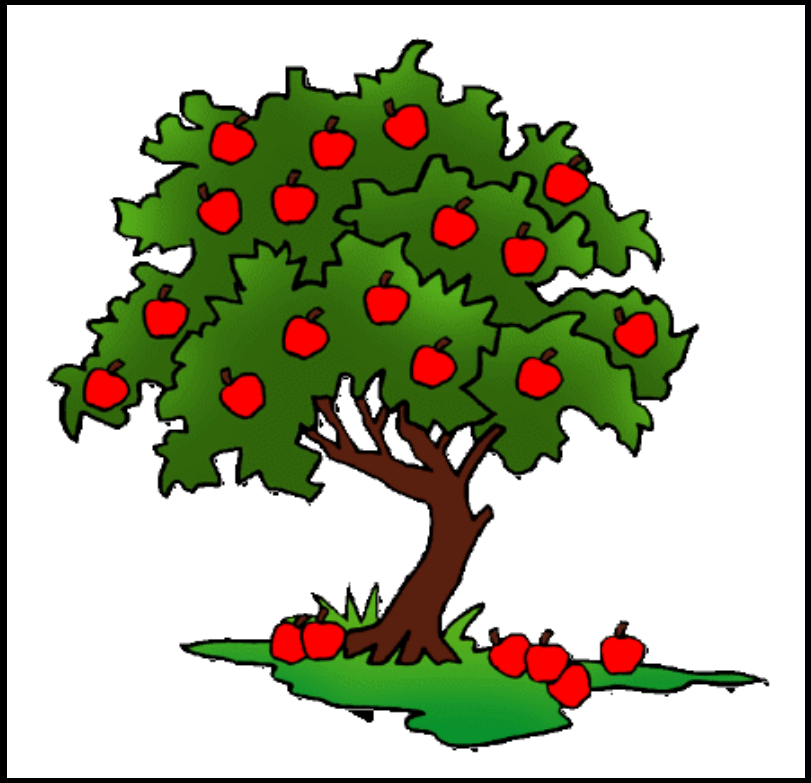
- Application 2: Matrix Multiplication
  - Introduction and Design (15 min)
  - Preparation (5 min)
    - Installing a skeleton code, compile test
  - Hands-on Programming (40 min)
    - Replace ??? with your own CUDA code
- Conclusion

Parallel Programming



- Goal: You have 1 minute to pick 100 apples
- There are 10 apple trees
- And you are a team of 10 people

Parallel Programming

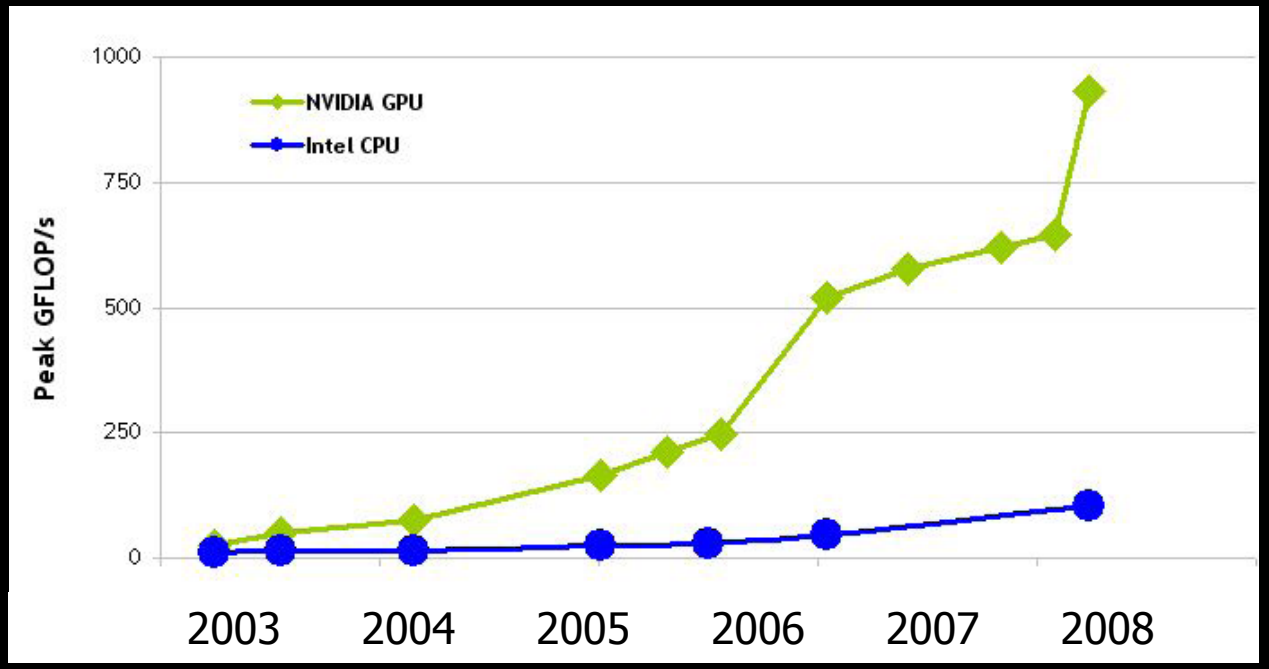


- You can assign each person to one apple tree
- Give them each a basket to collect the apples
- They may not be able to reach the apples at the top

Parallel Programming

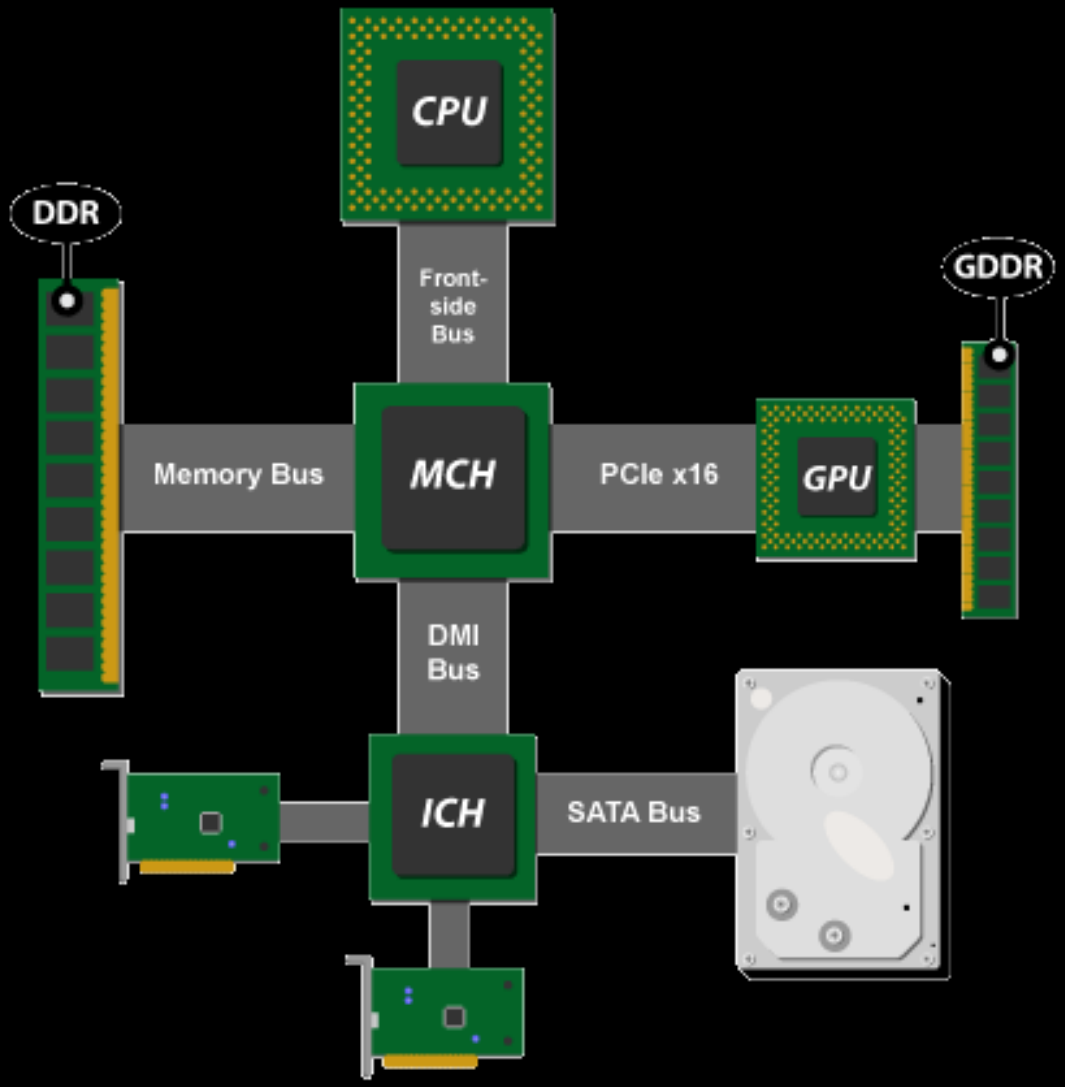
- Or you can assign two people to one apple tree!

Graphics Processing Units



FLOPS = **FL**oating point **O**perations **P**er **S**econd

The System

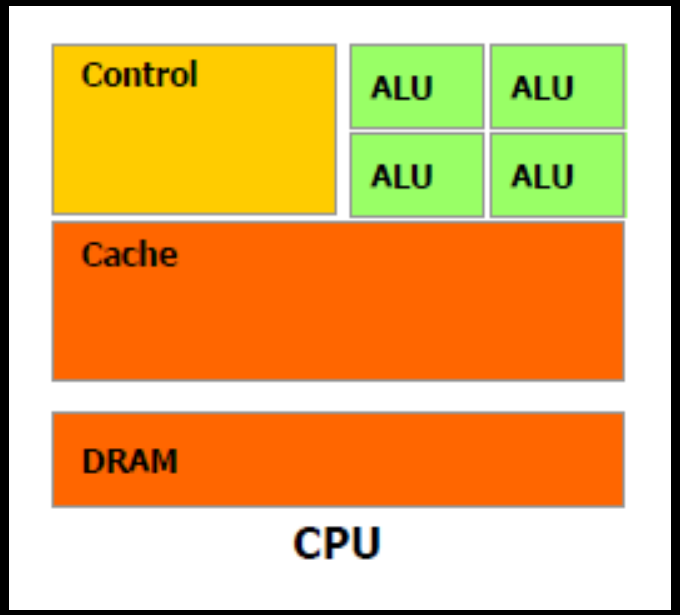


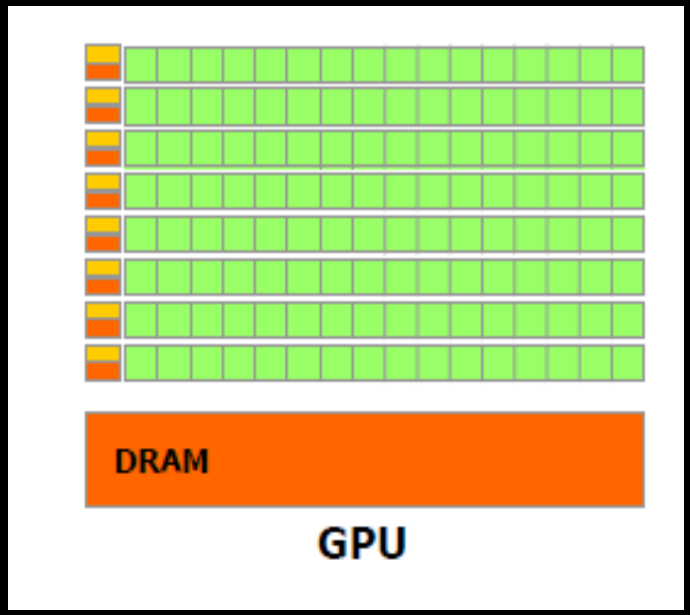
MCH = Memory Controller Hub

ICH = I/O Controller Hub

DDR = Double Data Rate

CPU vs GPU





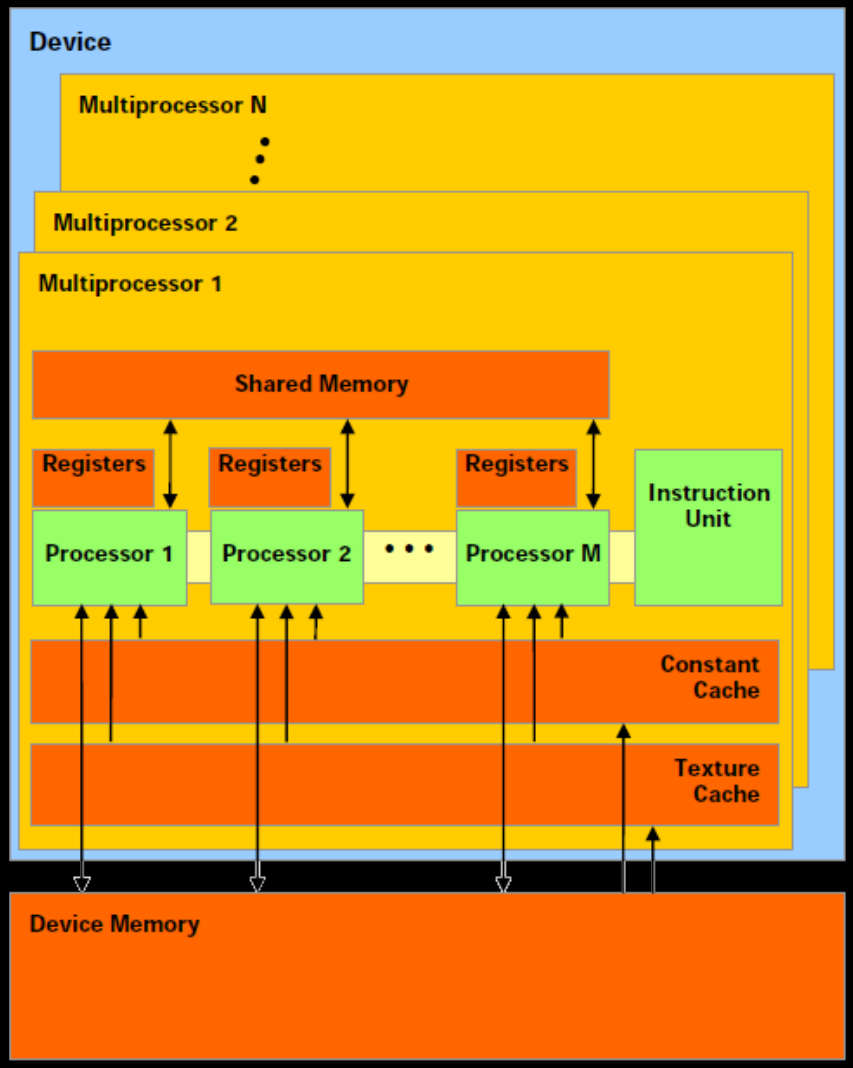
Irregular data accesses

Regular data accesses

ALU = Arithmetic Logic Unit
Cache: Like the apple basket

GPU Hardware Architecture

- Many cores
  - My laptop: 2 cores
  - Some of the newer desktops: 8 cores
  - GPUs: 16 x 768 = 12,288 cores!!!
- Many memory spaces
  - device (GDDR): Large but slow
  - shared (cache): Small but fast… this is the apple basket
  - constant and texture



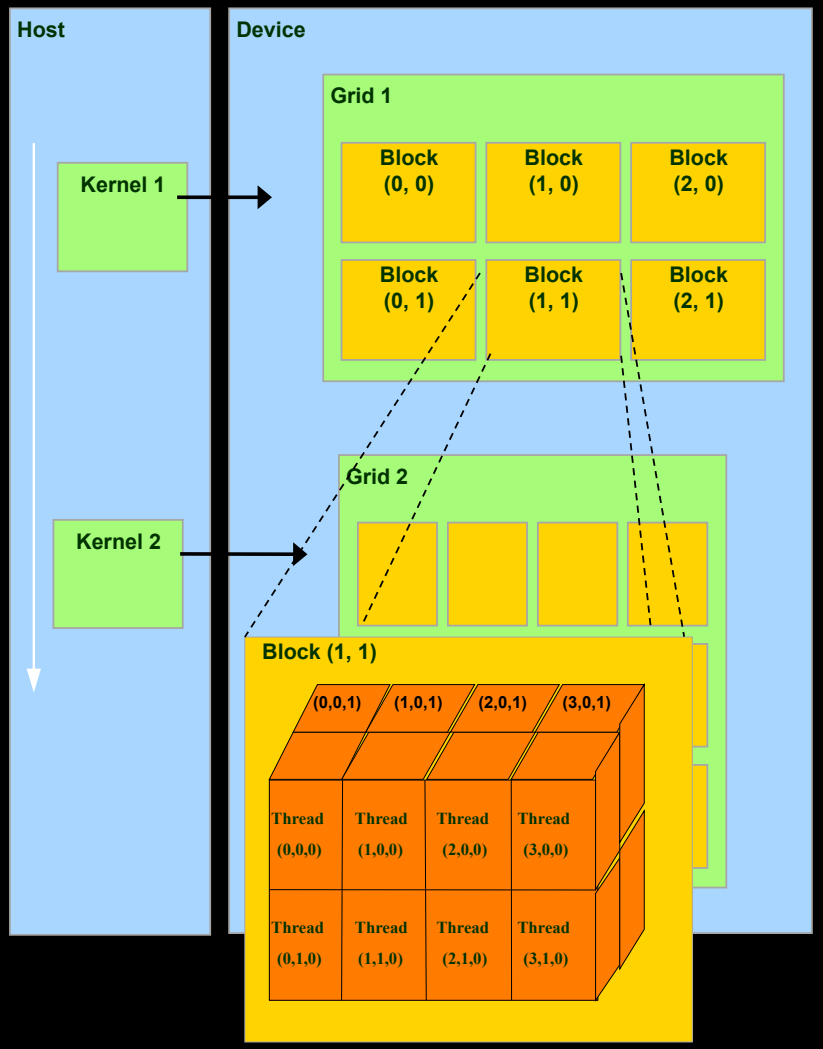
How to program GPUs

1. Transfer data from CPU to GPU
2. Decide how many threads (people) and how many groups (teams)
3. Write the GPU program (pick apples)
4. Transfer back the results from GPU to CPU

Thread Structure

CUDA terminology:

- The GPU program is called the **kernel**
- The kernel is executed by a **grid** of **threads**
- Threads are grouped into **blocks** which execute together on a core
- Each thread has a unique ID within the block
- Each block has a unique ID
- Threads within a block have access to common **shared** memory



Array Addition (CPU)

```
void arrayAdd(float *A, float *B, float *C, int N) {
   for(int i = 0; i < N; i++)
      C[i] = A[i] + B[i];
}
```

Computational kernel

```
int main() {

   int N = 4096;
   float *A = (float *)malloc(sizeof(float)*N);
   float *B = (float *)malloc(sizeof(float)*N);
   float *C = (float *)malloc(sizeof(float)*N);

   init(A); init(B);

   arrayAdd(A, B, C, N);

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

Allocate memory

Initialize memory

Deallocate memory

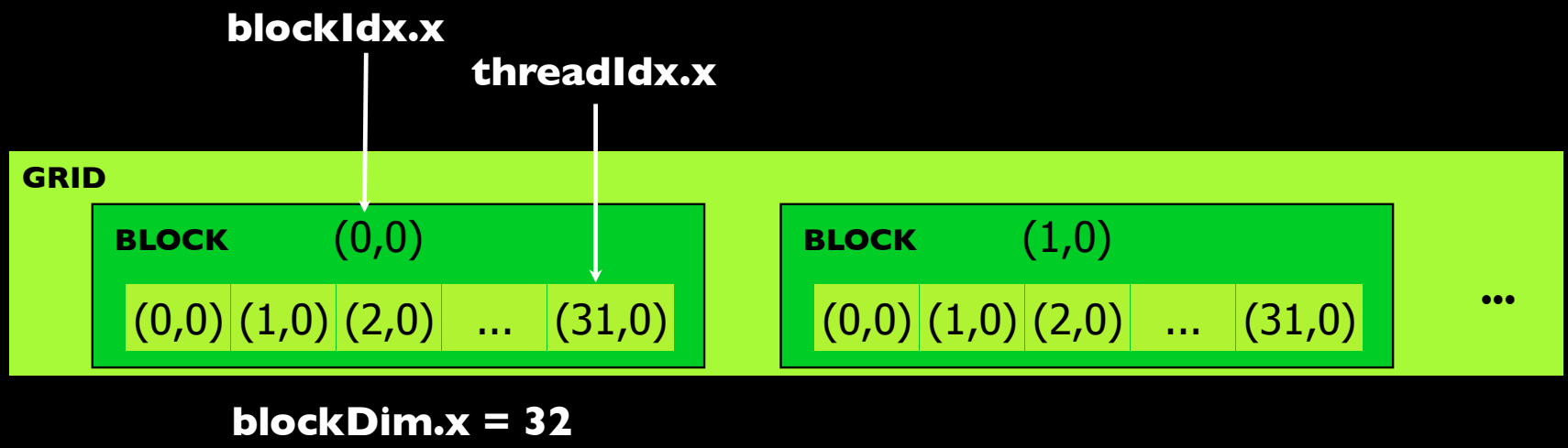
Array Addition (GPU)

```
__global__
void gpuArrayAdd(float *A, float *B, float *C) {

   int tid = blockIdx.x * blockDim.x + threadIdx.x;

   C[tid] = A[tid] + B[tid];
}
```

GPU Computational kernel



blockDim.x = 32

```
tid = blockIdx.x * blockDim.x + threadIdx.x
```

Vector Addition (GPU)

```
int main() {

   int N = 4096;
   float *A = (float *)malloc(sizeof(float)*N);
   float *B = (float *)malloc(sizeof(float)*N);
   float *C = (float *)malloc(sizeof(float)*N)

   init(A); init(B);

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

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

   dim3 dimBlock(32,1);
   dim3 dimGrid(N/32,1);

   gpuArrayAdd <<< dimBlock,dimGrid >>> (d_A, d_B, d_C);

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

   cudaFree(d_A);
   cudaFree(d_B);
   cudaFree(d_C);

   free(A); free(B); free(C);
```

← Allocate memory on GPU

← Initialize memory on GPU

← Configure threads

← Run kernel (on GPU)

← Copy results back to CPU

← Deallocate memory on GPU

Application 1: Image Rotation
- Introduction -

- Rotate an image by a given angle
- A basic feature in image processing applications

Original Input Image

Rotated Output Image

Application 1: Image Rotation
- Introduction -

- What the application does:

  Step 1. Compute a new location according to the rotation angle
  (trigonometric computation)

  Step 2. Read the pixel value of original location

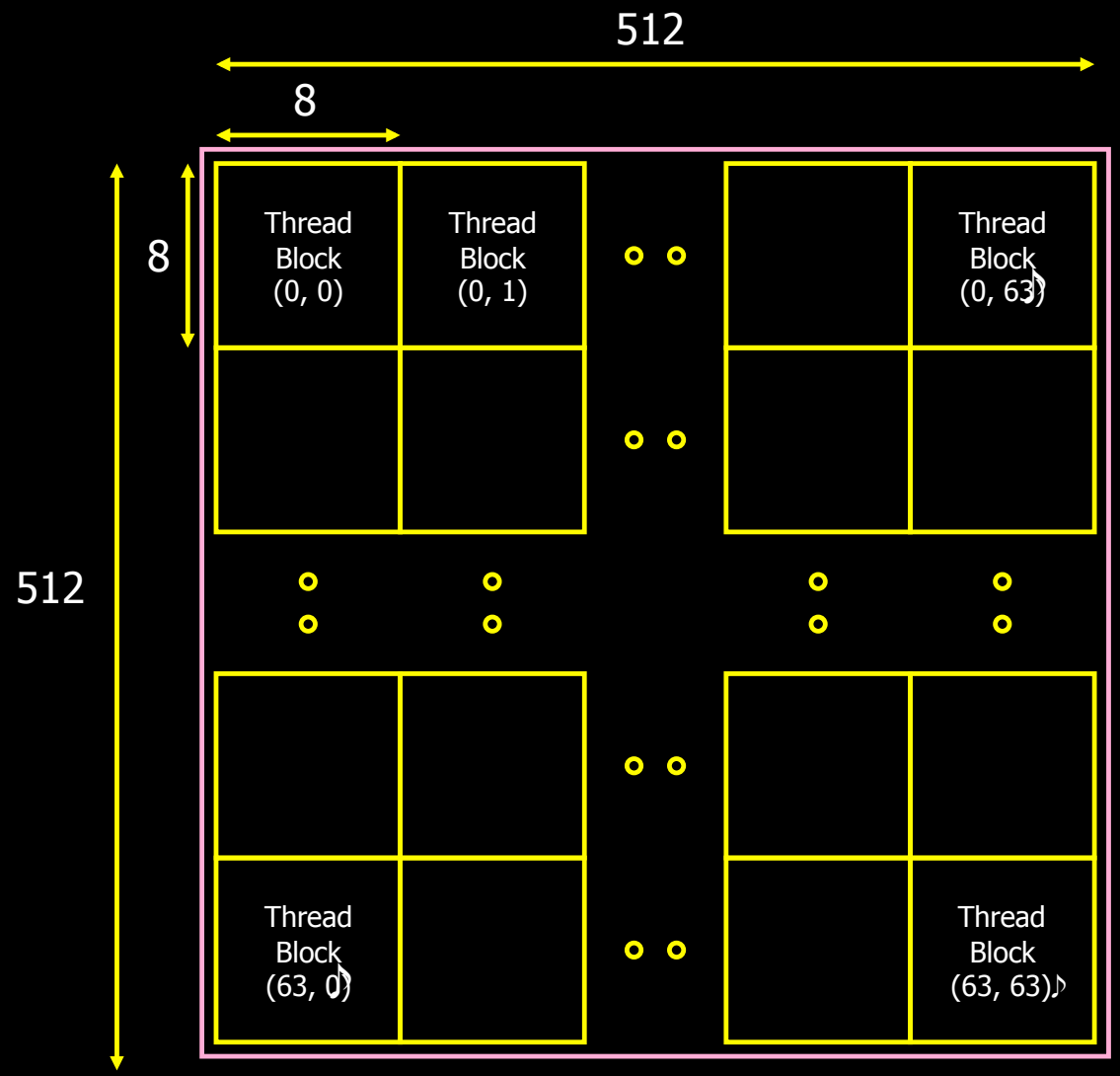
  Step 3. Write the pixel value to the new location computed at Step 1

- Create the same number of threads as the number of pixels
- Each thread takes care of moving one pixel
- Our goals are
  - To understand how to use GPU for data parallelism
  - To know how to map threads to data

Application 1: Image Rotation

- Design -

Threads Mapping



Application 1: Image Rotation
- Preparation -

1. Deploy the skeleton code in the proper directory

   [..@compute-0-8]$ tar xvf TeamProjects.tar

2. Compile

   [..@compute-0-8]$ cd Projects/cuda/src/ImageRotation/

   [..@compute-0-8]$ make clean

   [..@compute-0-8]$ make

   To use printf() to debug, use "make emu=1" instead of "make"

4. Execute

   [..@compute-0-8]$ ../../bin/linux/release/ImageRotation

5. Convert image from "pgm" to "jpg" format

   [..@compute-0-8]$ convert data/RadHouse_out.pgm data/RadHouse_out.jpg

6. Download "RadHouse_out.jpg" to your workstation to view it

Application 1: Image Rotation
- Hands-on Programming -

- Replace ??? in the skeleton code with your own CUDA code
- Refer to the hints and comments in skeleton code
- Talk to me if you have any questions or are done
- Try to finish by 6:30 pm
- Help others if you finish early

Application 2: Matrix Multiplication

- Introduction -

$$A_{HA,WA} \times B_{HB,WB} = C_{HC,WC}$$

$$\begin{bmatrix} a_{0,0} & \\ & \\ a_{HA,0} & a_{HA,WA} \end{bmatrix} \times \begin{bmatrix} b_{0,0} & \\ & \\ b_{HB,0} & b_{HB,WB} \end{bmatrix} = \begin{bmatrix} c_{0,0} & \\ & \\ c_{HC,0} & a_{HC,WC} \end{bmatrix}$$

Application 2: Matrix Multiplication
- Introduction -

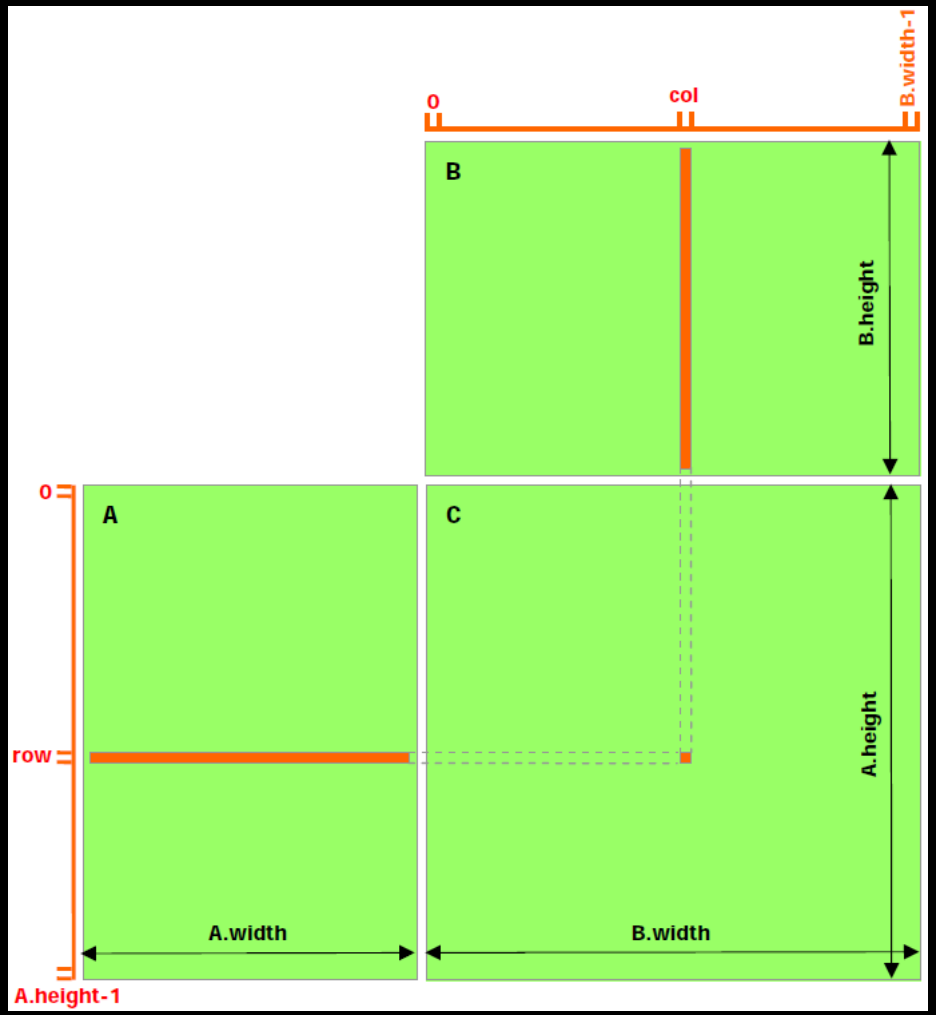
- Serial implementation looks like

```
for (int i=0; i < HC; i++)
        for (int j=0; i < WC; j++)
                for (int k=0; i < WA; k++)
                        C[i][j] += A[i][k] * B[k][j];
```

- Calculating C[i][j] happens in parallel
- We will use a fast shared memory to store per-block matrices (As and Bs) because shared memory is faster

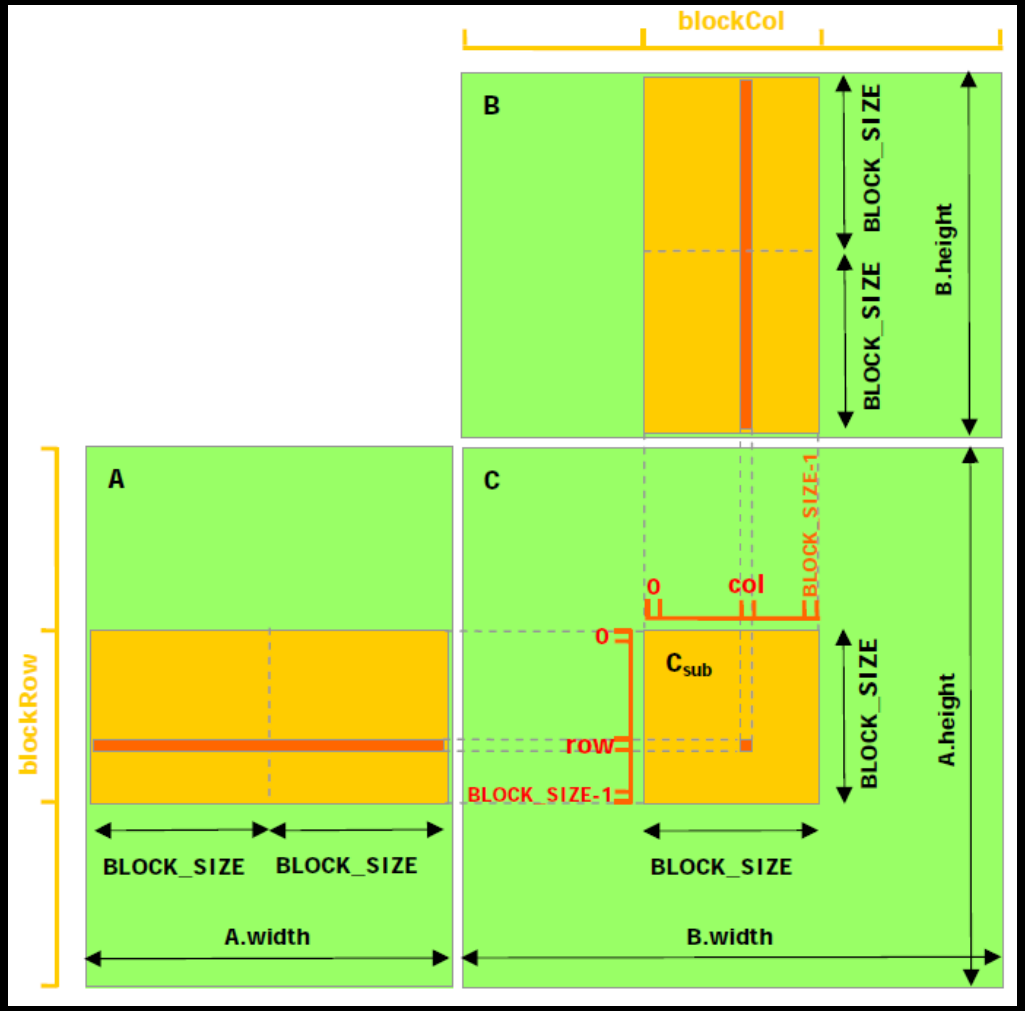
Application 2: Matrix Multiplication
- Design -

- Matrix multiplication without using shared memory



Application 2: Matrix Multiplication
- Design -

- Matrix multiplication using shared memory



Application 2: Matrix Multiplication
- Preparation -

1. Compile

   [..@compute-0-8]$ cd Projects/cuda/src/MatrixMul

   [..@compute-0-8]$ make clean

   [..@compute-0-8]$ make

   To use printf() to debug, use "make emu=1" instead of "make"

2. Execute

   [..@ac ~]$ ../../bin/linux/release/MatrixMul

4. Check output message

   "*** TEST FAILED": something wrong

   "*** TEST PASSED": you got it

Application 2: Matrix Multiplication
- Hands-on Programming -

- Replace ??? in the skeleton code with your own CUDA code
- Refer to the hints and comments in skeleton code
- Talk to me if you have any questions or are done
- Try to finish by 8:00 pm
- Help others if you finish early

Conclusions

- What we've learned throughout the two projects
  - Understood a massive parallel computing on GPU
  - Experienced what CUDA programming looks like
  - Understood how to explicitly program hardware resources
  - Understood the importance and challenges in parallel programming
  - Experienced solving problem in massively parallel fashion
- GPU is the platform of choice for data-parallel computationally-intensive applications
- In a few years, we are likely to see many people buying a new graphics card to increase the desktop's computing performance, not to increase 3D game performance
- What if my GPU is not CUDA-compatible? OpenCL!

More information

- NVIDIA GPU Computing Developer Home Page
  http://developer.nvidia.com/object/gpucomputing.html
- CUDA Download
  http://developer.nvidia.com/object/cuda_2_3_downloads.html
- Khronos OpenCL
  http://www.khronos.org/opencl/
- Programming Massively Parallel Processors: A Hands-on Approach, David B. Kirk and Wen-mei W. Hwu

Thank you!