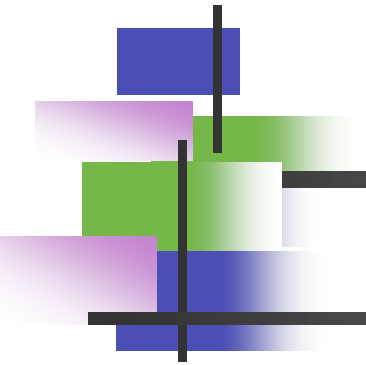


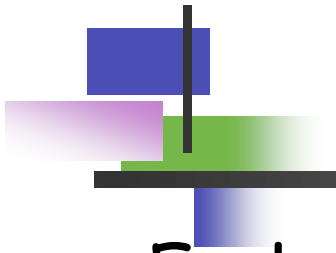
# 8. GPU Computing

Parallel Programming

Dr Hamidreza Khaleghzadeh
School of Computing
University of Portsmouth

# Goals

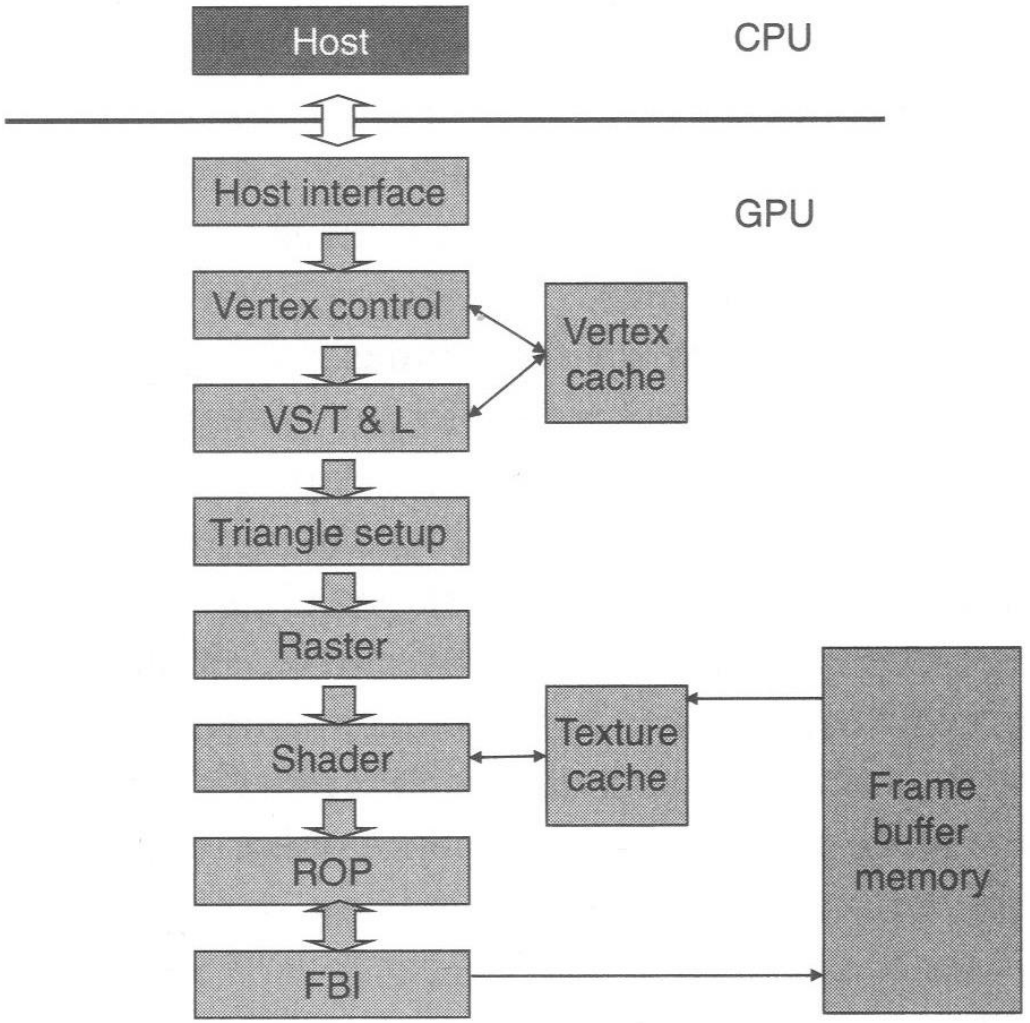
- Review historical evolution of *Graphics Processing Units*, towards "general purpose" massively parallel processors.
- Introduce *CUDA* programming model, with motivating example of matrix multiplication.
- Presentation closely follows the book by Kirk and Hwu.



# GPUs and Real-Time 3D Graphics

- Early *Graphics Processing Units* consisted of fixed function pipelines
  - *Configurable* but not *programmable*
- 3D images built of from *polygons* – typically triangles.
- *Vertex* is one corner of a triangle or polygon.

# Fixed Function Graphics Pipeline†

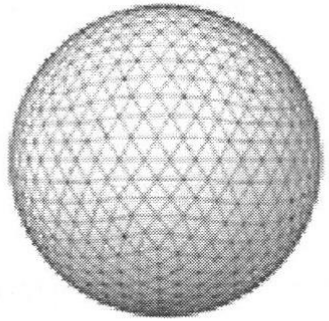



†Kirk & Hwu, PMPP, Fig 2.1

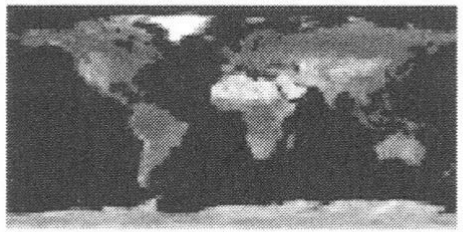
# Stages in Pipeline

- GPU receives triangle data from Host
- *Vertex control* converts vertices to some internal representation (in *vertex cache*).
- *Vertex shading, transform and lighting* assigns colours, texture coordinates, etc to *vertices*.
- *Triangle setup* creates *edges* of triangles.
- *Raster* stage determines pixels in each triangle, and values used for shading pixels.
- *Shader* colours pixels, e.g. applying texture
- *Raster Operation* stage deals with overlapping triangles – occlusion, etc.

# Application of Texture†

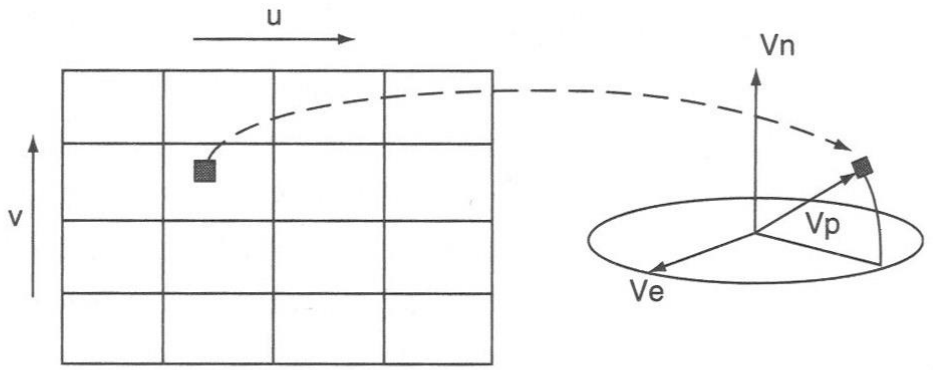


Sphere with no texture



Texture image

Sphere with texture

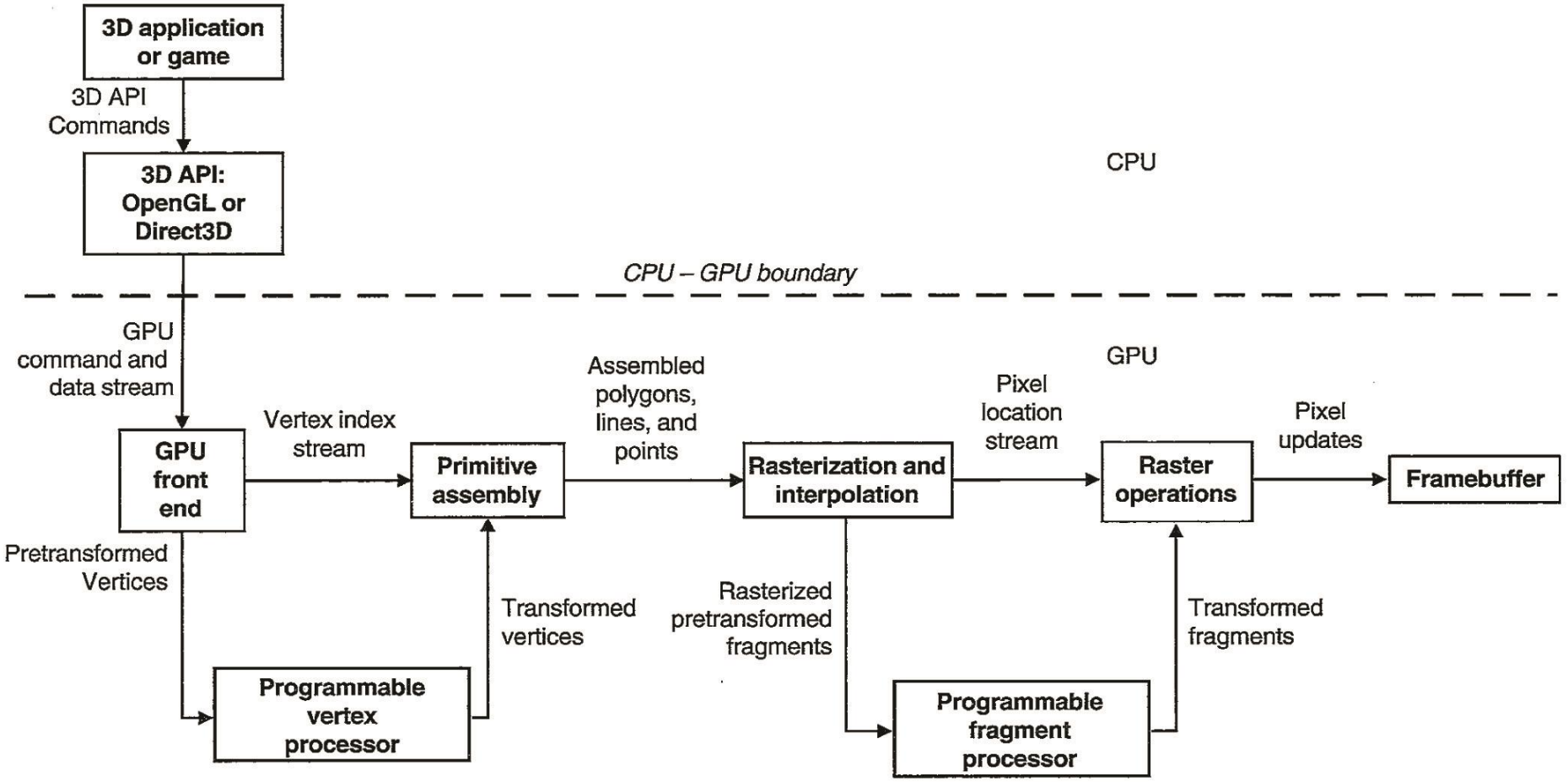



†Kirk & Hwu, PMPP, Fig 2.2

# Programmable Pipelines

- As application developers demanded increasingly sophisticated effects, parts of the pipeline became programmable
- In the architecture on the next slide, the *VS/T & L* stage (aka the *vertex shader*) and the fragment (pixel) processing from the erstwhile *shader* stage have become programmable.

# A Programmable Pipeline†

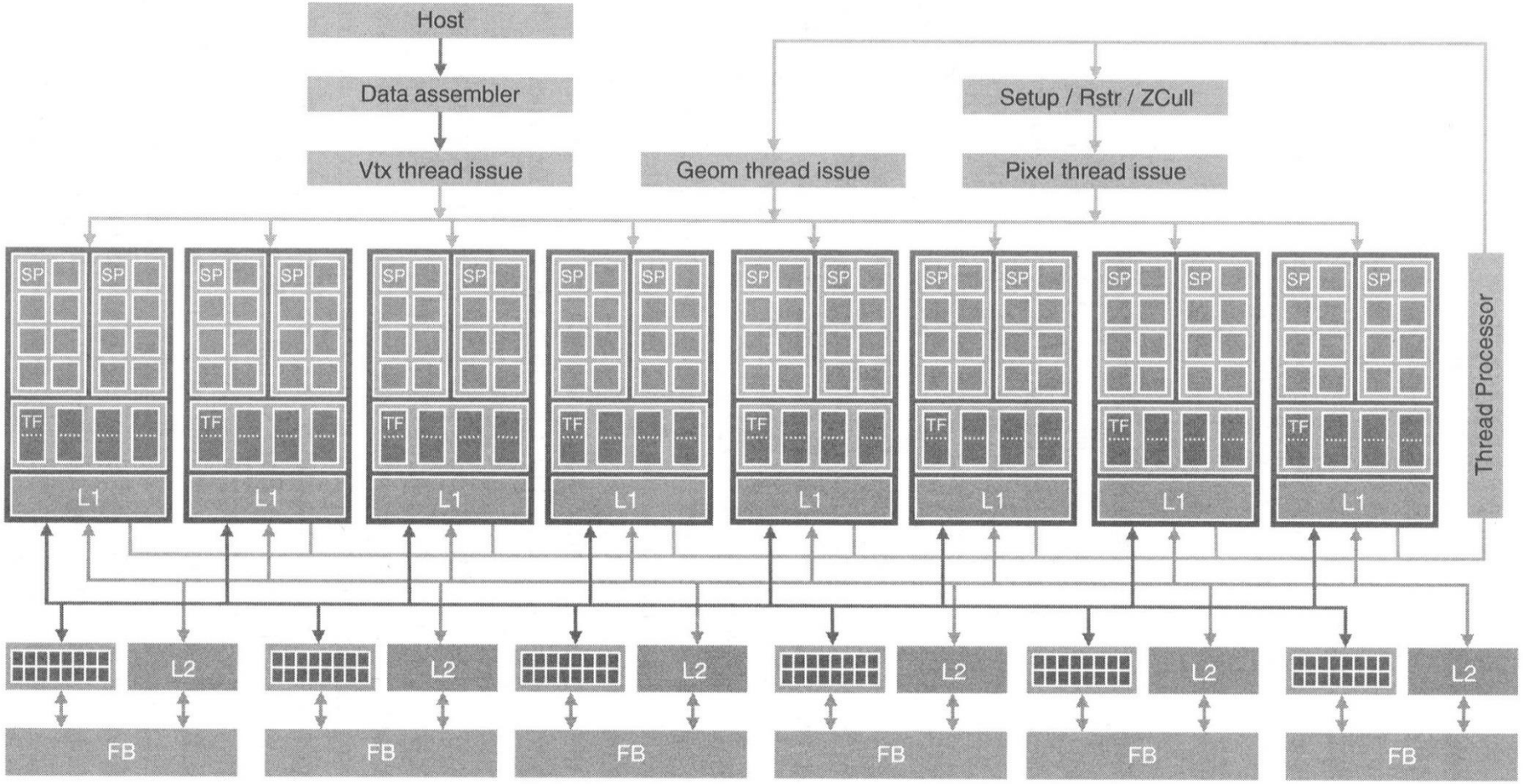



†Kirk & Hwu, PMPP, Fig 2.4

# Unified Programmable GPUs

- What happened next - as demands increased and efficient utilization of the silicon became important - was that it became unattractive to have *dedicated* units for the separate programmable stages.
- Still many fixed function stages, but user programmable logic was unified into one massively parallel array of processing elements with load balancing, so that the main pipeline could "recirculate" through the programmable array in its different programmable stages.

# GeForce 8800 GTX Pipeline†

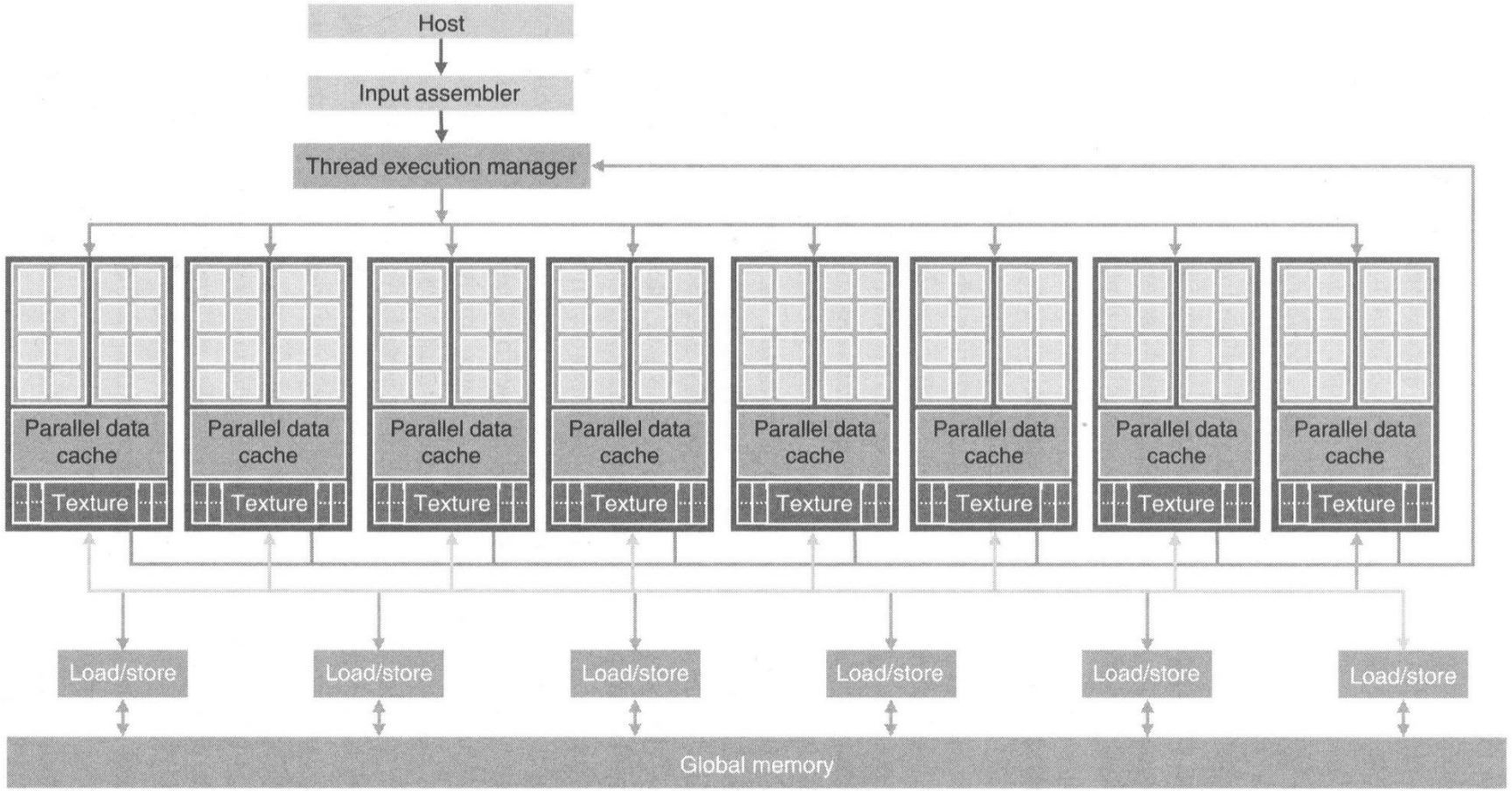



†Kirk & Hwu, PMPP, Fig 2.5

# GPU Computing

- Designers of this generation of GPUs consciously added features that would be particularly valuable to non-graphics programmers – e.g. the ability to write to arbitrary locations in memory (instead of just writing a fixed pixel in the frame buffer).
- Circa 2006, GPU computing came of age...

# GTX Viewed More Abstractly†




†Kirk & Hwu, PMPP, Fig 1.3

# CUDA

# Graphics APIs

- The incremental increases in programmability of the GPUs had been driven by increasing functionality demanded by APIs used by games and graphics programmers
  - *Direct3D* – part of Microsoft *Direct X*
  - *OpenGL* – an open standard API
    - Supports programmable shaders written in e.g. *GLSL*.
- Some researchers had noticed that such APIs could be coerced into use for non-graphics applications, and the first generation of "*General Purpose computing on GPUs*" (*GPGPU*) exploited the APIs.

# Arrival of CUDA

- In 2007 NVIDIA released the *CUDA* Software Development Kit.
- This provided *a dialect of the C/C++ programming language* that could be used for programming compatible GPUs as "general purpose" massively parallel processors.
- We will follow Kirk and Hwu in introducing CUDA through a *matrix multiplication* example.
- But first we need some general background on matrices and arrays in C.

# ASIDE: MATRICES AND ARRAYS

# Matrices as 2D Arrays

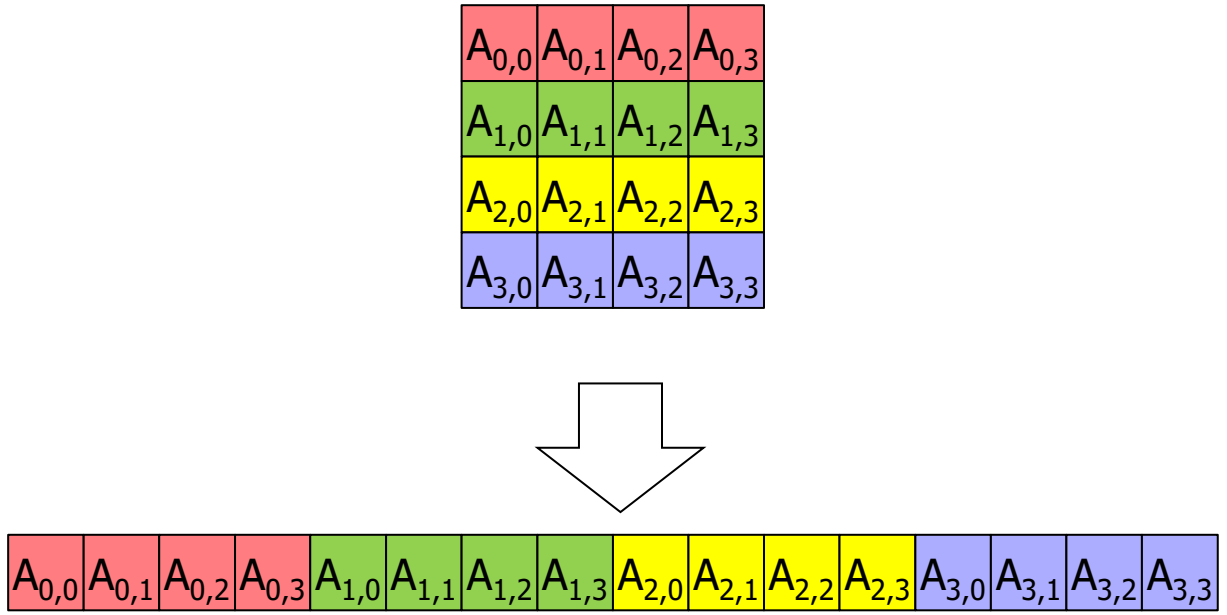
- Most obvious way to represent a matrix is as a 2 dimensional array.
- So, for example if *A* and *B* are two matrices size *N* by *N* and we want their product, *C* :

```
for (int i = 0 ; i < N ; i++)
    for (int j = 0 ; j < N ; j++) {
        C [i] [j] = 0.0
        for (int k = 0 ; k < N ; k++)
            C [i] [j] += A [i] [k] * B [k] [j] ;
    }
```

# 2D Data Structures in C

- Like most high level languages, *C* has 2D arrays, but they can be inflexible to use
  - Their *type signatures* have *compile-time constant* dimensions.
- Quite often when we have two dimensional data structures like matrices in *C* and other languages, we *flatten* them and store them in *one-dimensional arrays*.
  - (Although 2D arrays in Java are more flexible, they have their own overheads, and it isn't unusual to do similar tricks there.)

# Mapping a 2D Array to a 1D Array



# Matrix Multiplication Revisited

- Now suppose *A*, *B* and *C* are represented as one-dimensional arrays of size $N^2$.
- Matrix multiplication can be written like this:

```
for (int i = 0 ; i < N ; i++)
   for (int j = 0 ; j < N ; j++) {
      C [N*i + j] = 0.0
      for (int k = 0 ; k < N ; k++)
         C [N*i + j] += A [N*i + k] * B [N*k + j] ;
   }
```

# Arrays and Pointers in C

- In the C programs below, all arrays will be arrays of floating point numbers, and the type signature used in declaring these arrays will be float*, e.g.:

```
void matmul(float* A, float* B, float* C, int N)
```

  - Technically, float* means a *pointer to a variable of type float*.
  - But, in C, arrays are represented as pointers, and for our purposes you can just think of float* as the type of a one-dimensional array of floating point numbers.

# DATA PARALLEL MATRIX MULTIPLICATION

# Massive Multi-Threading

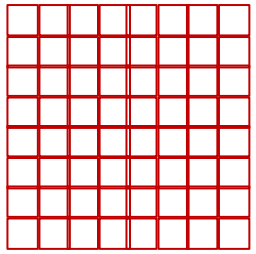
- In parallel programming multicore CPUs, we typically used a number of threads similar to the number of *cores* (e.g., around four).
- A GPU may have hundreds or even thousands of *processing elements*.
- CUDA adopts a "thread-oriented" model of programming the GPU, but generally encourages the use of *many more threads* than the number of available processing elements.
  - So *at least* thousands of threads.

# Memory Bandwidth and Latency

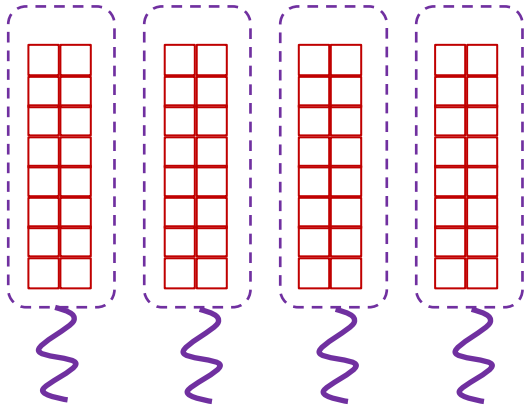
- GPUs typically have very high *memory bandwidth* compared with CPUs (i.e. they can transfer many GBs per second).
- But they also have high *latency* for accessing memory (individual *small* transfers takes longer).
  - Tradeoffs driven by original use of GPUs for graphics.
- Running *many threads per PE* helps to *hide the latency* – while one thread is waiting for memory, another can get on with useful work.

# Massive Multi-threading Illustration

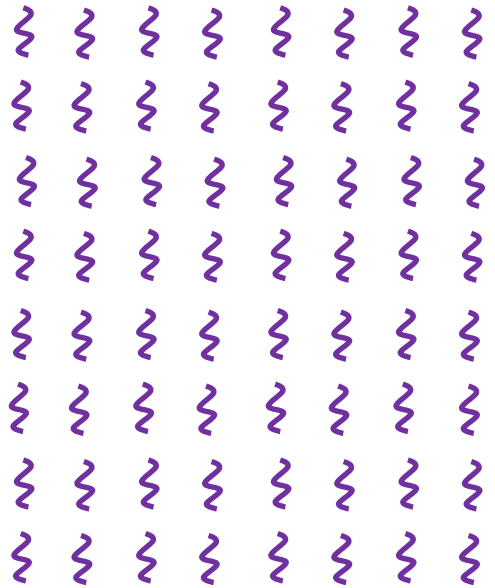
*N* by *N* data array



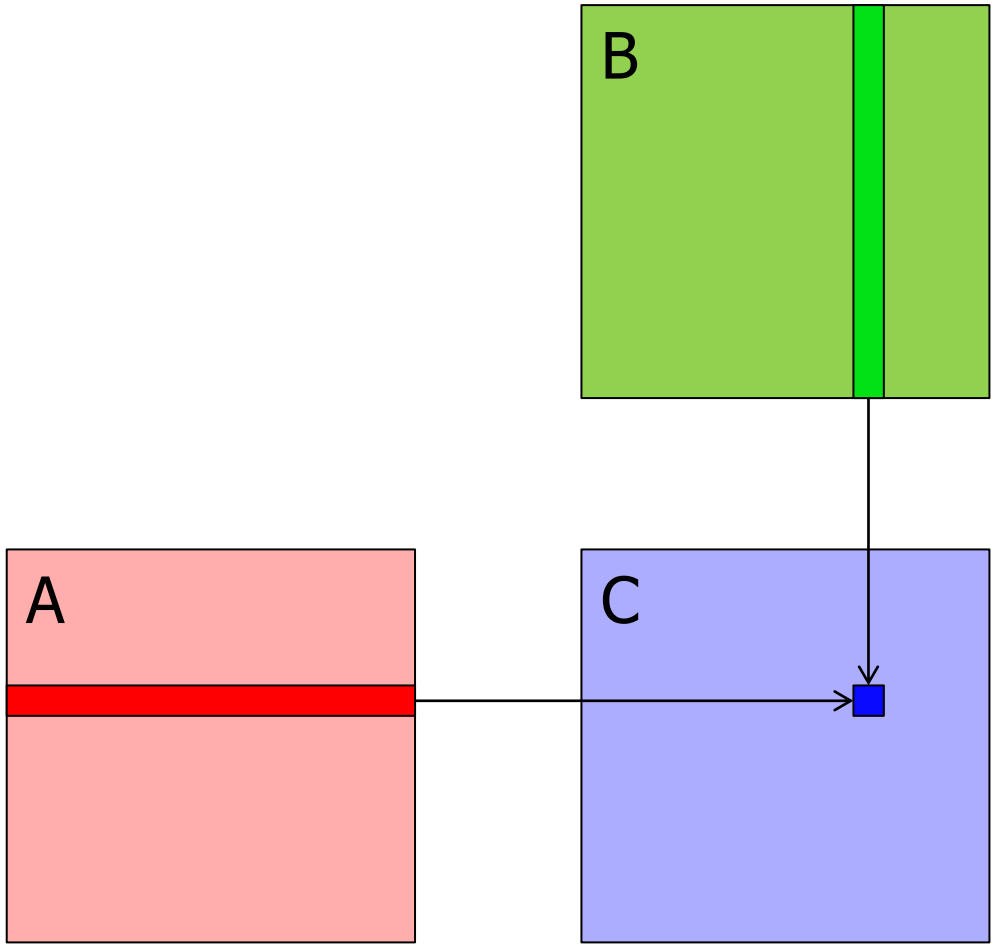
Multicore with threads – *P* threads:



GPU – *N* by *N* threads:



# Visualization of Matrix Multiplication



# Data Parallelism

- Each element of *C* is calculated independently as a dot product of one row of *A* with one column of *B*.
- All these elements can be calculated independently, so there is a massive amount of *data parallelism* in the problem – essentially $N^2$ tasks all involving work proportional to *N*.

# A CUDA "Thread"

```
__global__  void matmulKernel(float* Ad, float* Bd,
                              float* Cd, int N) {
   int tx = threadIdx.x ;   // 2D Thread ID
   int ty = threadIdx.y ;

   float sum = 0 ;
   for(int k = 0 ; k < N ; k++) {
        sum += Ad [ty * N + k] * Bd [k * N + tx] ;
   }
   Cd [ty * N + tx] = sum ;
}
```

# Comments

- The __global__ keyword before the function definition identifies this as a *kernel function* – one that runs on the GPU, but is effectively "called" from the host CPU.
- This thread calculates *one element of C*.
- The special variables threadIdx.x, threadIdx.y play a role similar to our familiar me variable, but CUDA supports virtual *2D arrays of threads* – so these are effectively just *i*, *j*.
- *Ad*, *Bd*, *Cd* are *almost A*, *B*, *C*...

# CUDA Host Program

```
void matmul(float* A, float* B, float* C, int N) {
    ... create arrays in device memory (see following slides) ...
    ... copy host arrays A and B to device arrays Ad and Bd ...

    // Set up the execution configuration
    dim3 dimBlock(N, N) ;
    dim3 dimGrid(1, 1) ;

    // Launch the device computation threads!
    matmulKernel<<<dimGrid, dimBlock>>>(Ad, Bd, Cd, N) ;

    ... copy device array Cd to host array C (see later) ...
}
```

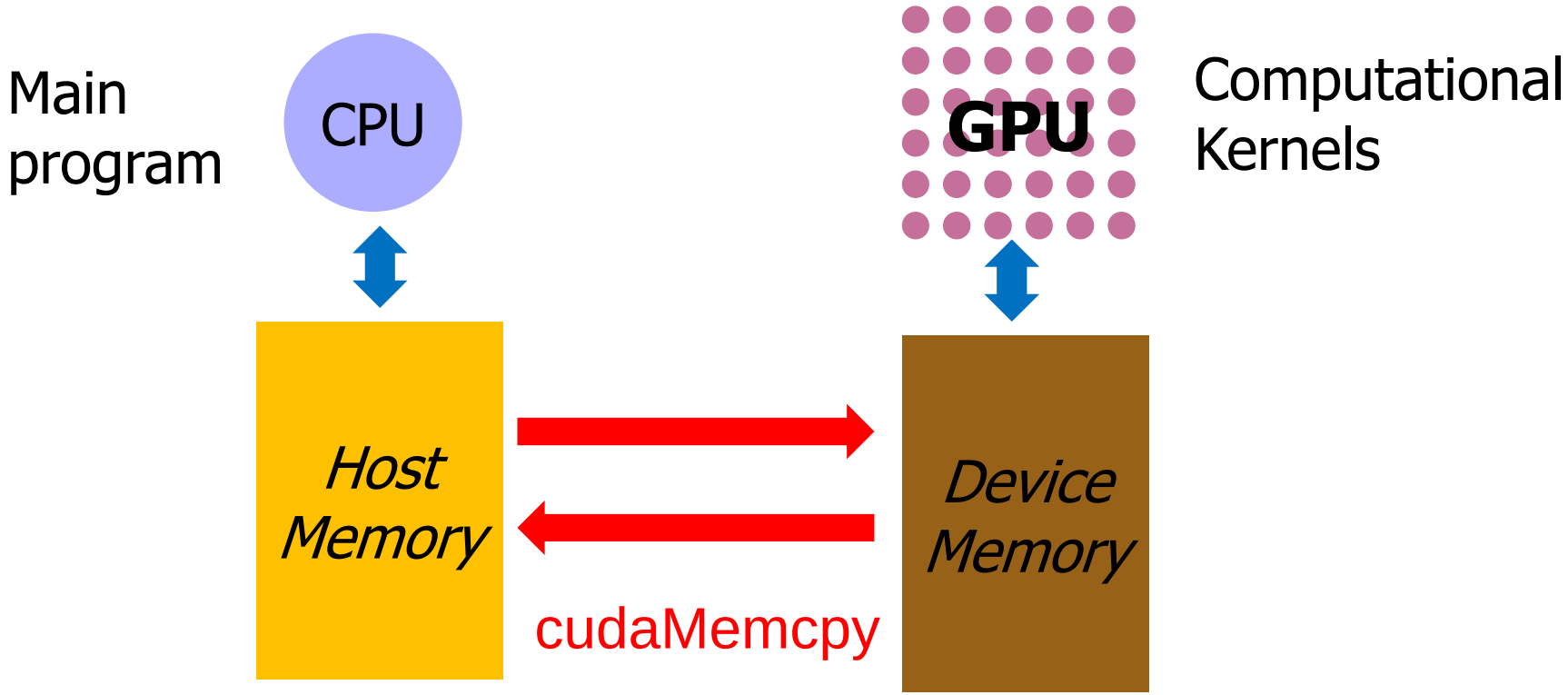
# Notes

- We request an *N* by *N* block of threads by creating a suitable dim3 object (see later for discussion of blocks)
- Then we just call the kernel function that runs on the device in *N* by *N* separate threads.

# Device Memory

- In CUDA terminology, the *device* is the GPU, the *host* is the CPU it is attached to.
- The GPU has *its own external, fast memory*, *distinct* from the CPU *main memory*. A kernel function can only directly access this device memory.
- A feature of CUDA (and its main competitor *OpenCL*) is that the programmer must *explicitly manage the transfer* of data between the host and device memories.

# Host and Device Memory



# Creating arrays in device memory

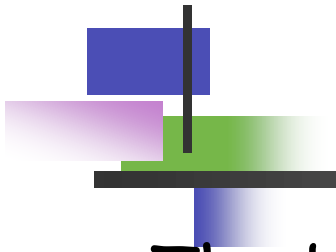
```
void matmul(float* A, float* B, float* C, int N) {
    // First, create three arrays in device memory
    float *Ad, *Bd, *Cd ;
    int size = N * N * sizeof(float) ;  // in bytes – C!
    cudaMalloc((void**) &Ad, size) ;
    cudaMalloc((void**) &Bd, size) ;
    cudaMalloc((void**) &Cd, size) ;
    ... Do the rest (see next slide) ...
}
```

# … The Rest …

```
void matmul(float* A, float* B, float* C, int N) {
    ... create arrays in device memory (see previous slide) ...
    // Copy host arrays A and B to device arrays Ad and Bd
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice) ;
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice) ;
    ... Launch threads on GPU (see slide 30) ...
    // Copy device array Cd to host array C
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost) ;
}
```

# Notes

- cudaMalloc is the CUDA function that allocates memory for arrays in the *GPU memory* – it follows the fairly clunky pattern of standard C malloc. Think of it as something like new.
- cudaMemcpy copies values in a host array to a device array, or vice versa, depending on the value of the 4$^{th}$ argument.



# Summary

- The basic programming *CUDA* model is quite straightforward, and ingenious as far as it goes.
- But… we have skipped over various complications in GPU programming introduced by specialized nature of hardware – more on these next week.
- Next week will also discuss other frameworks and applications for GPUs.

# Further Reading

- David B. Kirk and Wen-mei W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 2nd edition, Morgan Kaufman, 2013 (*PMPP*).