GPGPUs

Sandro Spina
Computer Graphics and Simulation Group

Computer Science Department
University of Malta

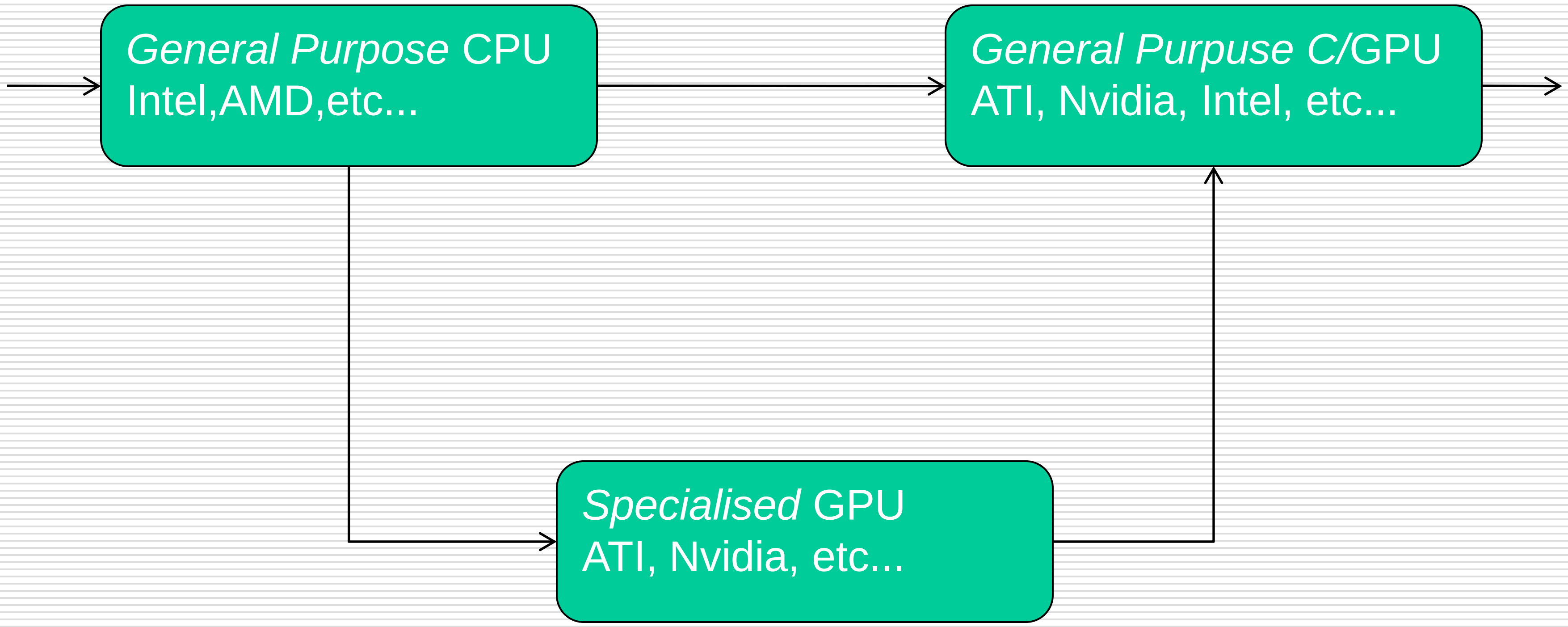
GPU Computing – A Short History of ...

- For over 3 decades computer scientists/engineers have been working hard on methods intended to re-create the 3D-world around us on a 2D screen.
- This manipulation of 3D models inside a computer requires a huge number of mathematical calculations ...
- ... and the need to update in real-time (30 frames per second) means that operations must occur very rapidly
- To address these unique 3D computational requirements specialised equipment (GPUs) with enormous number-crunching capabilities were developed

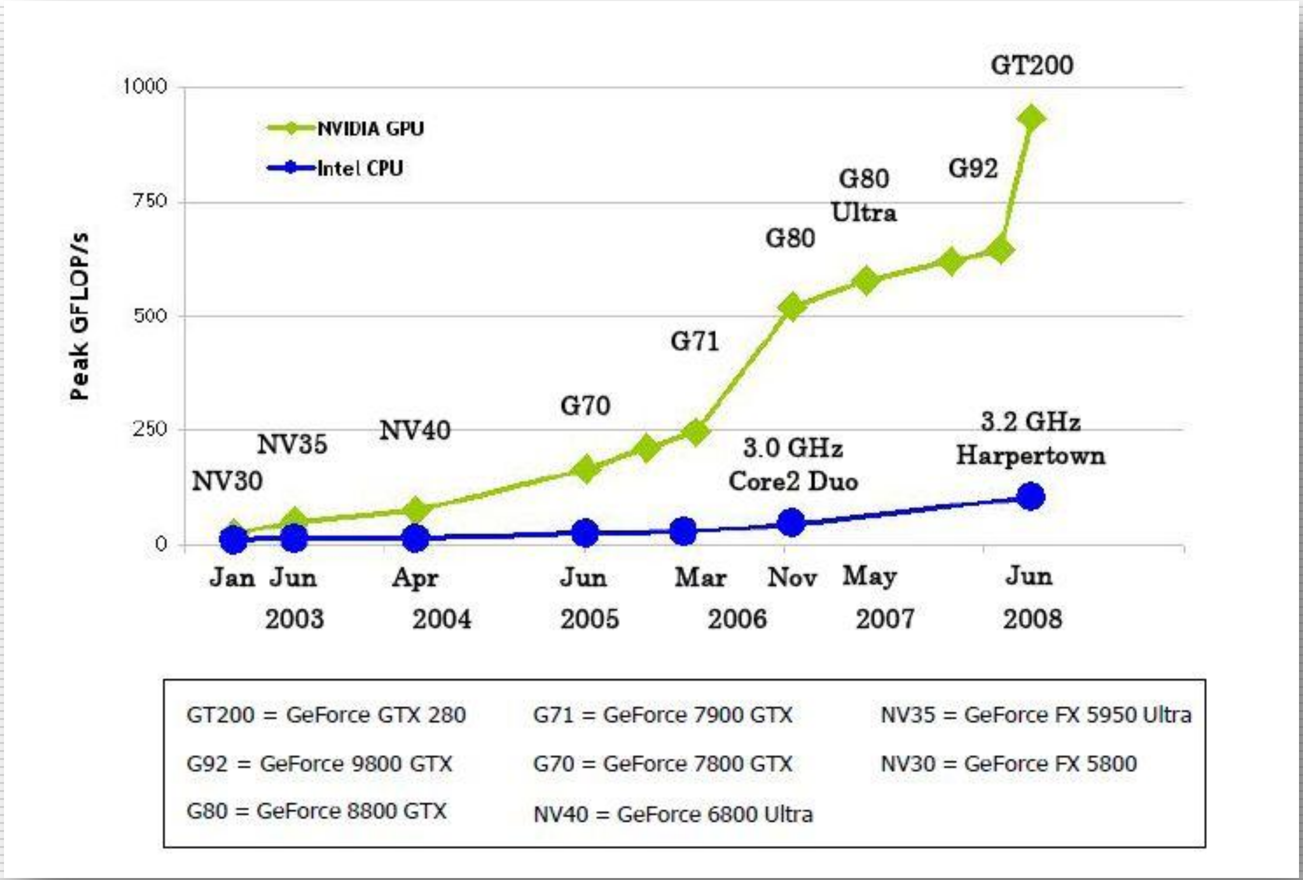
GPU Computing – A Short History of ...

- Over the years GPUs continued to advance ... These chips were optimised to handle the floating-point matrix math used to manipulate and render objects inside the computer’s 3D subsystem.
- These improvements can be witnessed just by looking at the current generation of computer games. At how these are becoming ever more realistic in terms of geometry detail and lighting calculations.
- As GPUs continued to bulk up their number-crunching capabilities in the pursuit of increased 3D realism, a group of scientists saw an opportunity to tap these now impressively advanced capabilities for rapid calculations to accelerate the non-graphical application software.
- This insight saw the advent of General Purpose GPU (GPGPU) computing.

GPU Computing – A Short History of …



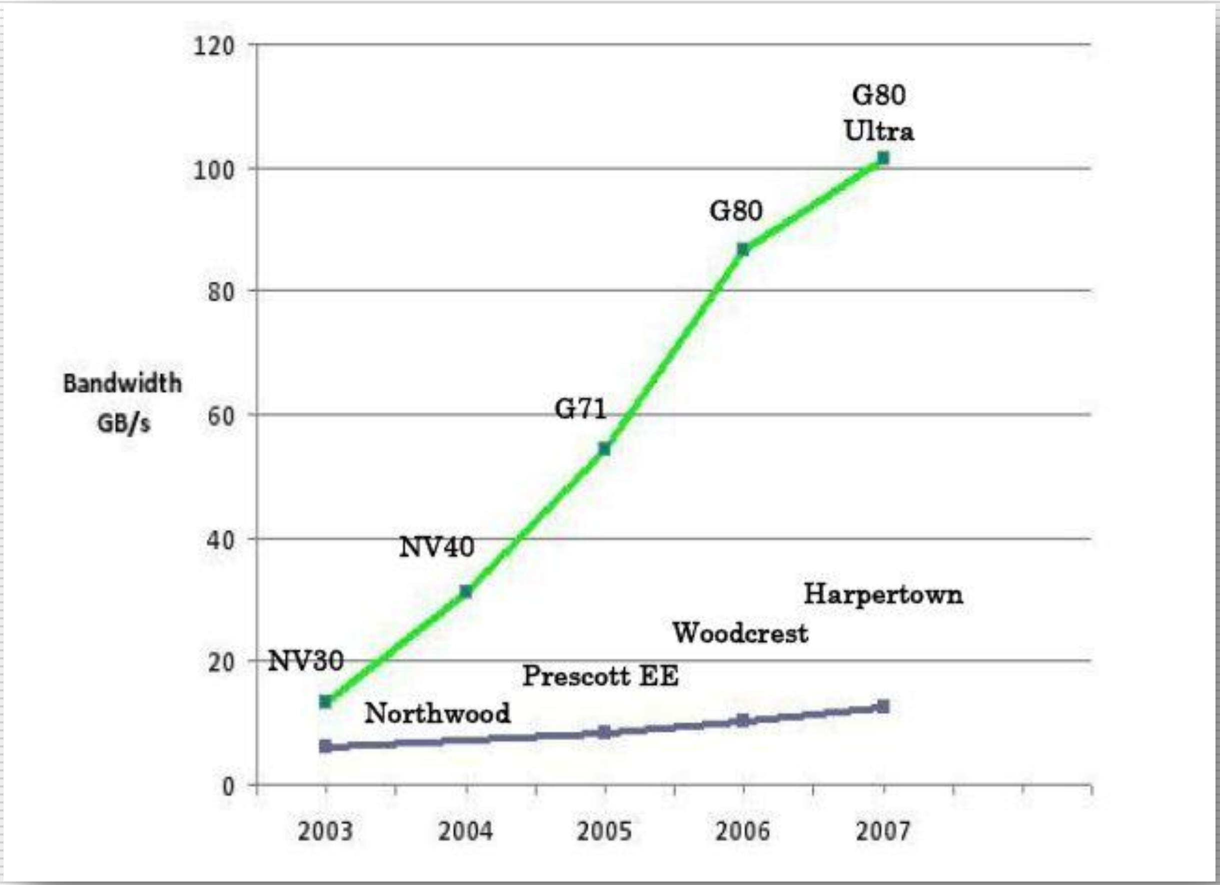
GPU vs CPU Comparison GLOPS/s



GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	

Source: NVIDIA_CUDA_Programming_Guide v2.0

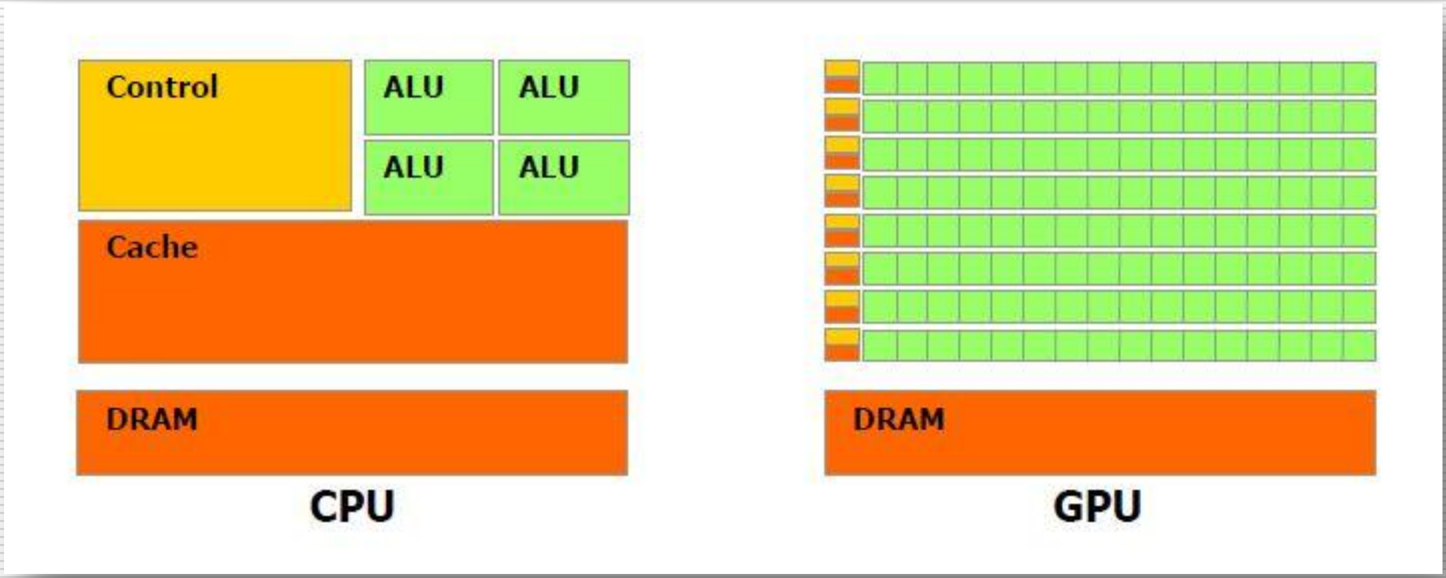
GPU vs CPU Bandwidth Comparison



Source: NVIDIA_CUDA_Programming_Guide v2.0

GPU Characteristics ….

The GPU is specialised for compute-intensive, highly parallel computation (necessary for real-time rendering) and therefore designed in such a way that more transistors are devoted to data processing rather than data caching and flow control ...



Source: NVIDIA_CUDA_Programming_Guide v2.0

Data Parallelism – S(I/P)MD model

- Shared Instruction (or Program) Multiple Data
- GPUs are highly optimised to address problems that can be expressed as data-parallel computations.
- i.e. The same instruction or program (chunk of instructions) is executed on many data elements in parallel.
  - Ex. The Map Operator in an array, Image Processing
- Data parallel processing maps data elements to parallel processing threads.
- Egs. Video Encoding, pattern recognition, physics simulations

NVIDIA Compute Unified Device Architecture – CUDA

- NVIDIA started off their effort into GPGPU computing with the introduction of the CUDA architecture in 2006.
- It first appeared with the launch of the GeForce 8800.
  - Simplifies the use of their GPUs for computational, as opposed to graphical applications
  - Provide libraries that allow programmers writing in C to access GPU features.
- The second iteration of CUDA
  - First appeared in the GTX200 series (GTX260/280)
  - Huge boost in performance
  - 64-bit floating point support
  - Broader set of software development tools to access these features
- Over the past two years, CUDA has evolved from a set of C extensions to a software and hardware abstraction layer that supports a host of programming languages and APIs (ex OpenCL, DirectCompute)

CUDA Abstractions …. For SIMD parallelism

- CUDA is essentially a **scalable** parallel programming model … Exposing three major abstractions:

A Hierarchy of Thread Groups

Shared Memories

Barrier Synchronisation

Kernels (CUDA and in general)

- *Kernels* are programs written specifically to be executed in parallel.
- ‘C for CUDA’ extends C by allowing the programmer to define kernels (C functions) that when called,
  - Are executed N times in parallel
  - By N different CUDA threads

CUDA Kernel definition and invocation

Definition:

```
__global__ void sum(float *a1, float *a2, float *a3)
{ … }
```

Invocation:

```
int main()
{ …
    sum <<<1,N>>> (a1,a2,a3);
… }
```

CUDA threadIdx

- Is a 3-component vector
- The intention here is to give easier access to 1,2 and 3 dimensional data

- 1 Dimension – Vector
- 2 Dimension – Matrix
- 3 dimension – Field

Threads, Blocks and Grids

Thread Hierarchy

- A set of threads make up a Block
- A set of blocks made up a Grid

- This hierarchy is also visible in other GPGPU languages which simply introduce a further layer of abstraction over the hardware.

- For example (as we shall see later) OpenCL uses the terms workItem and workGroup to address this thread hierarchy.

Threads, Blocks and Grids (Nvidia)

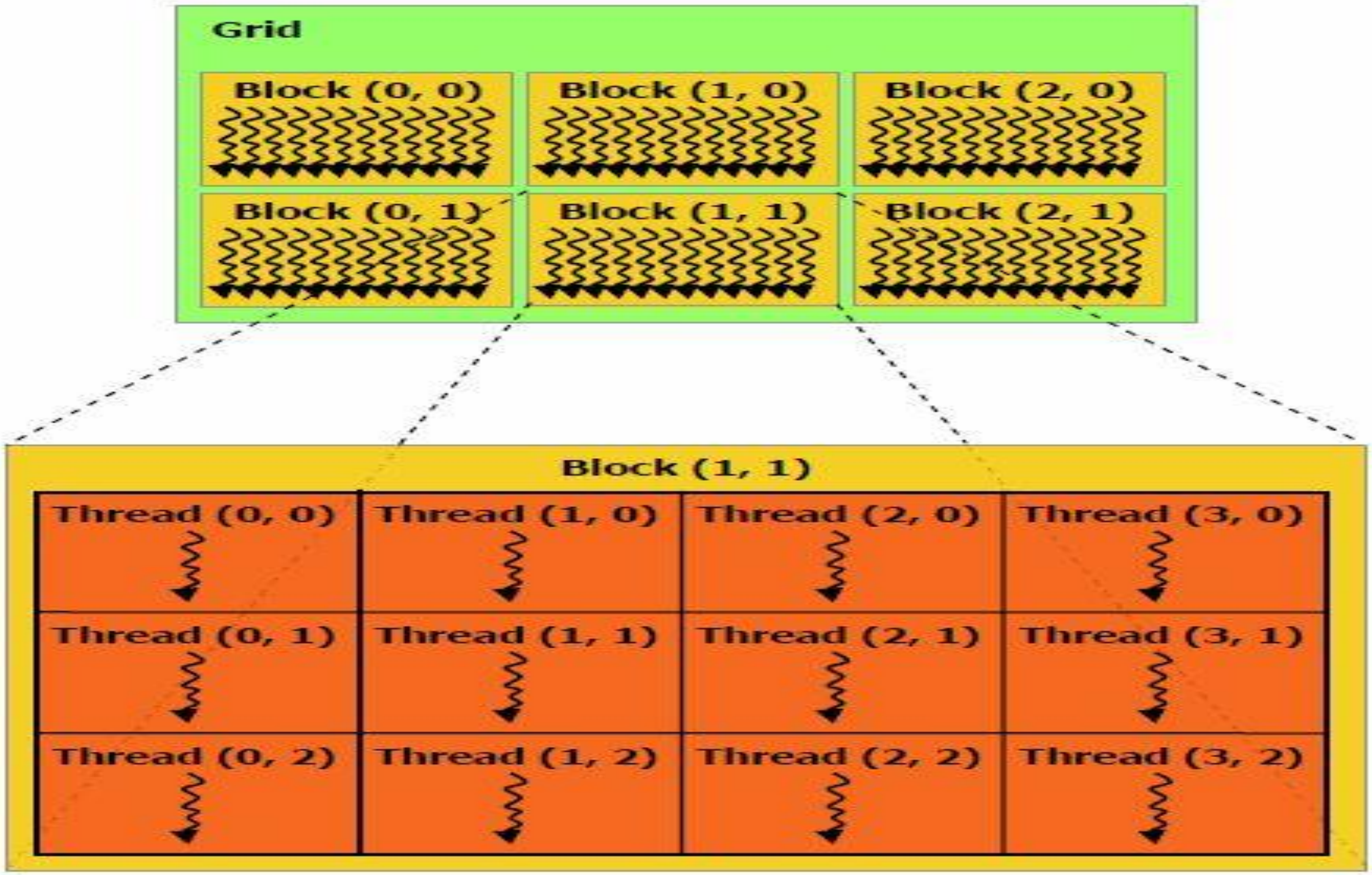


Figure 2-1. Grid of Thread Blocks

CUDA threadIdx

- Is a 3-component vector
- The intention here is to give easier access to 1,2 and 3 dimensional data
- 1 Dimension – Vector
- 2 Dimension – Matrix
- 3 dimension – Field

- Suppose you want to carry out matrix operations … you would use the 2D components of threadIDX + <<<1, (N,N)>>>

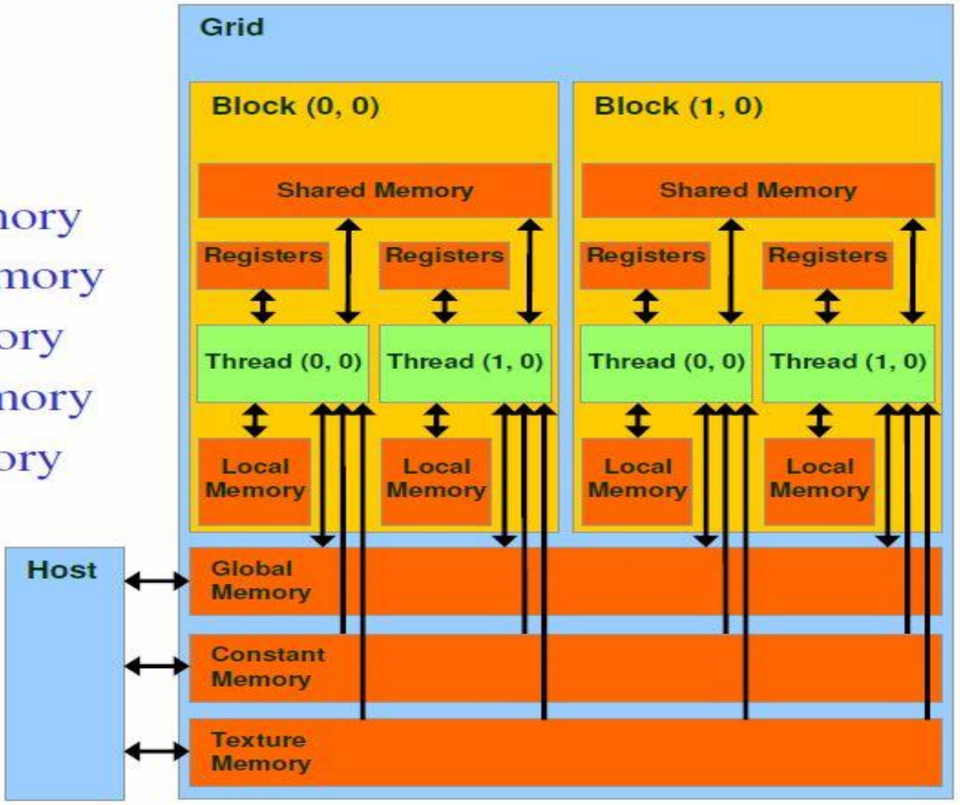
CUDA Memory Model

Programming Model: Memory Spaces

Each thread can:

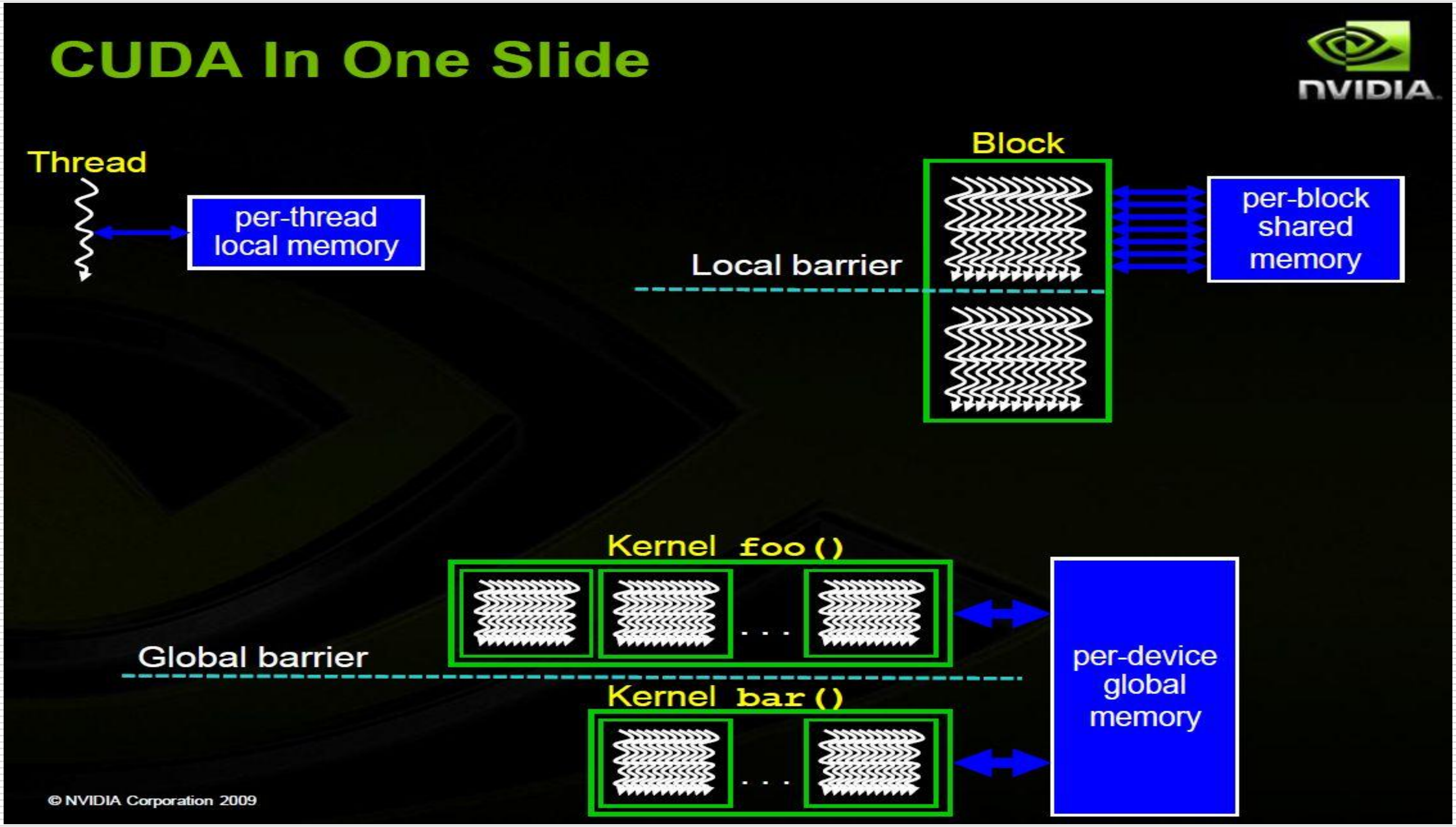
- Read/write per-thread registers
- Read/write per-thread local memory
- Read/write per-block shared memory
- Read/write per-grid global memory
- Read only per-grid constant memory
- Read only per-grid texture memory

The host can read/write global, constant, and texture memory



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

CUDA Memory Model (ii)

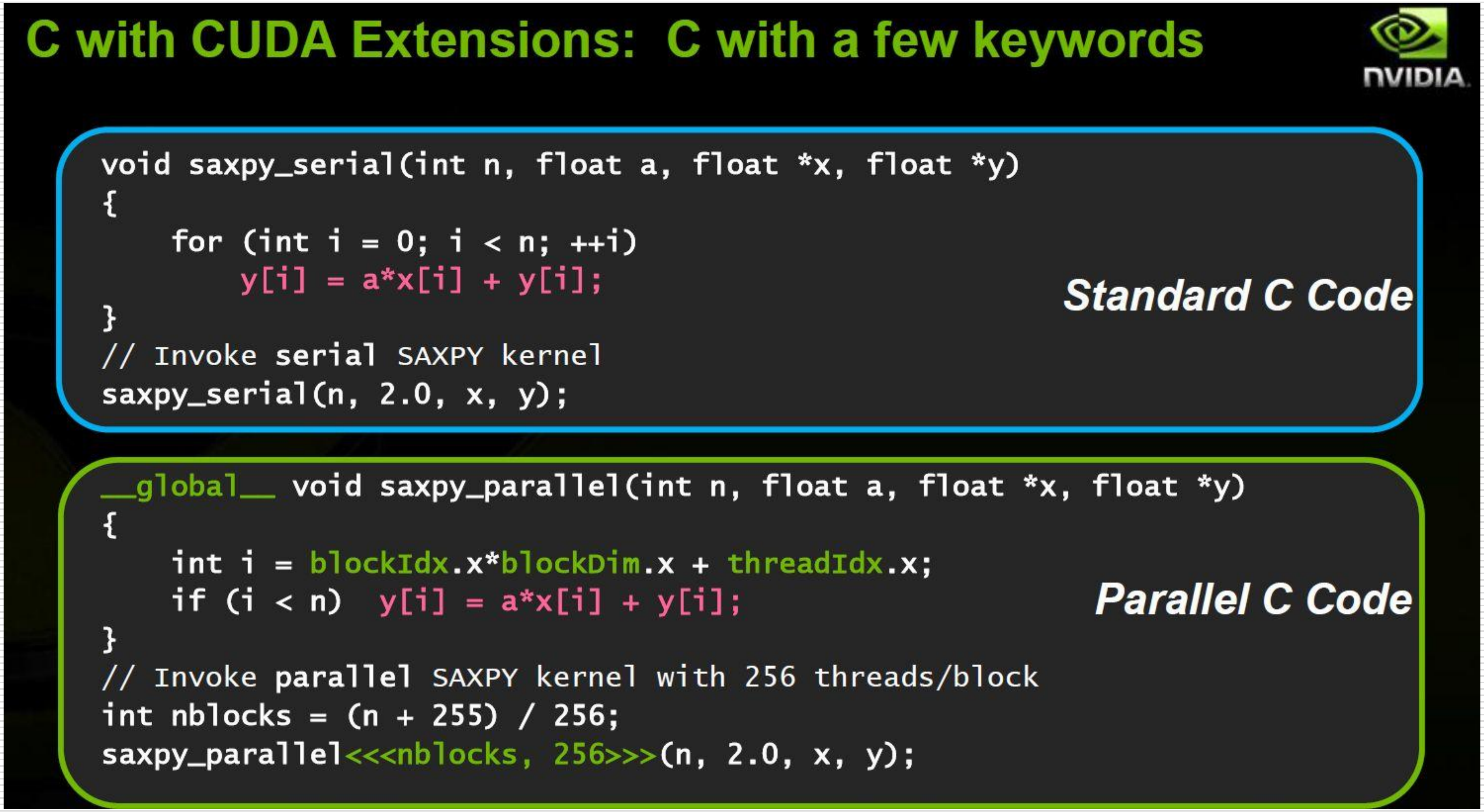


CUDA threadIdx Example 2D

```
__global__ void matAdd(float A[N][N], float B[N][N], float C[N][N])
{
        int i = threadIdx.x;
        int j = threadIdx.y;
        C[i][j] = A[i][j] + B[i][j];
}
int main()
{
// Kernel invocation
dim3 dimBlock(N, N);
matAdd <<<1, dimBlock>>>(A, B, C);
}
```

- Code Source: Nvidia CUDA Programming Guide v2.0

CUDA saxpy example …



C with CUDA Extensions: C with a few keywords

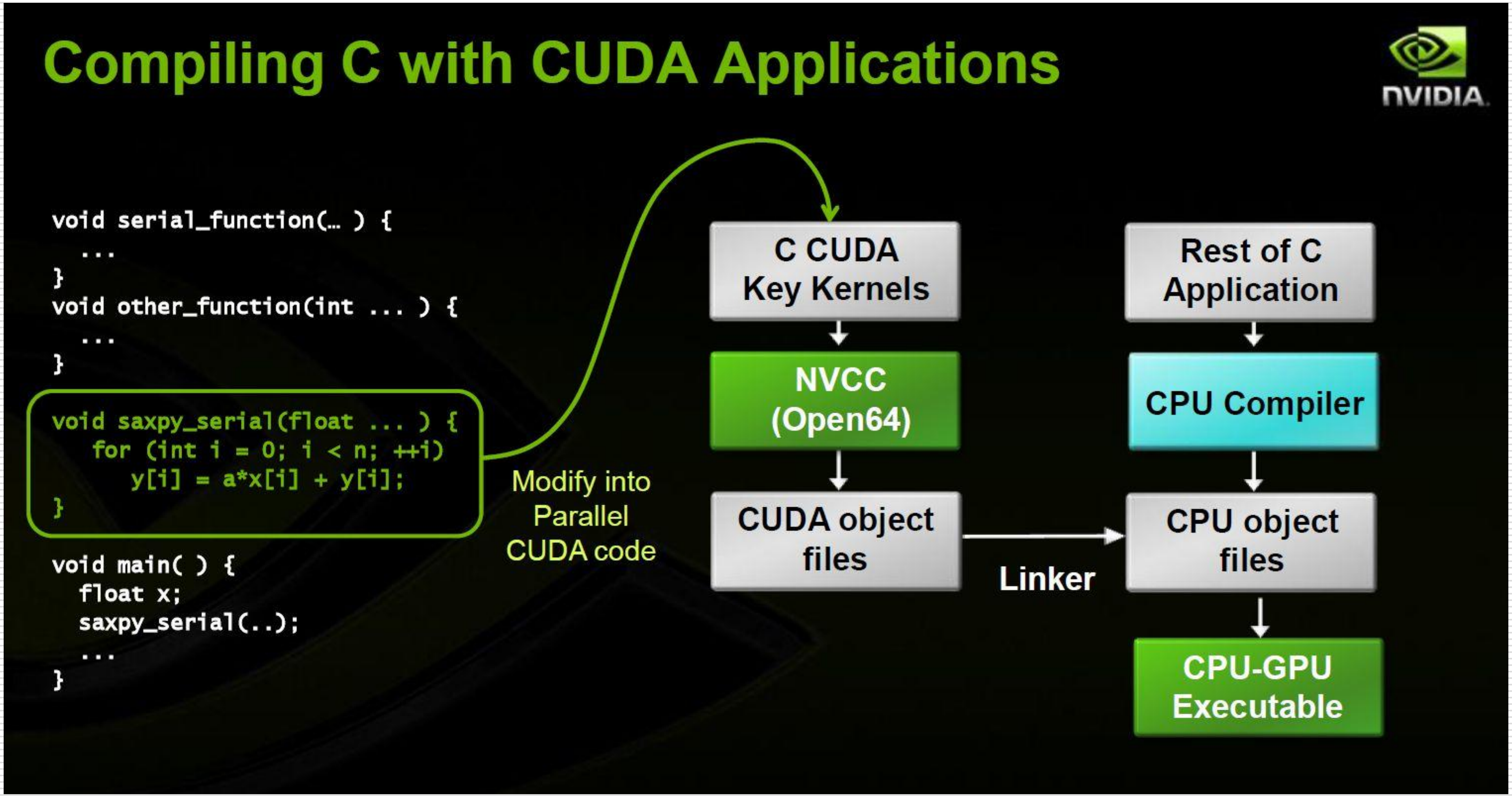
```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Standard C Code

```
__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)  y[i] = a*x[i] + y[i];
}
// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

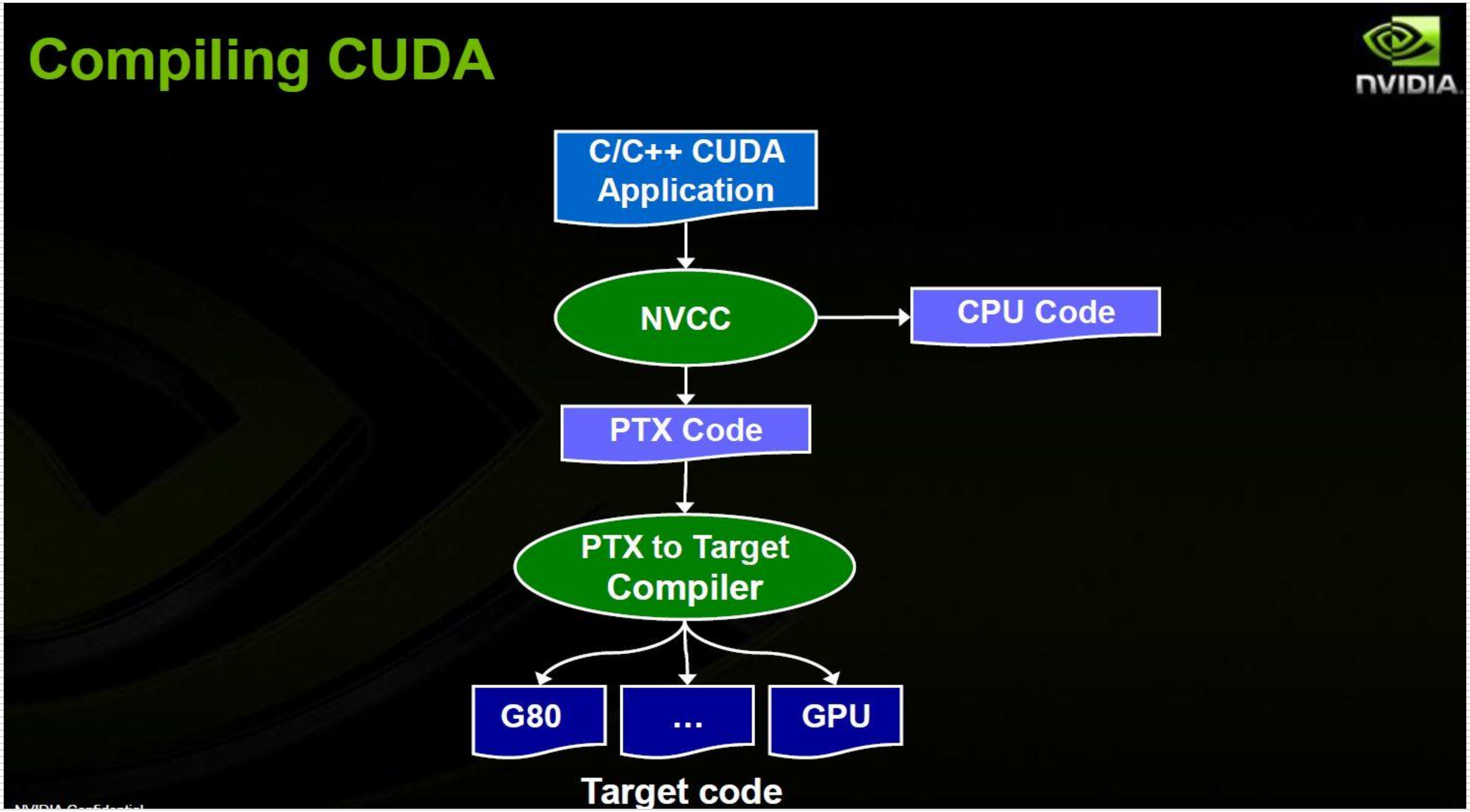
Parallel C Code

CUDA's Compilation Process



```
void serial_function(… ) {
   ...
}
void other_function(int ... ) {
   ...
}
void saxpy_serial(float ... ) {
   for (int i = 0; i < n; ++i)
      y[i] = a*x[i] + y[i];
}
void main( ) {
   float x;
   saxpy_serial(..);
   ...
}
```

CUDA's Compilation Process



CUDA Threads and Warps (i)

- Distinguish between CPU and GPU threads.
- The main difference lies in the thread scheduler ...
- CPU threads are executed independently ... Where this is not the case with GPU threads.
- GPU threads are scheduled in groups of ***warps***.
- The threads within a warp are executed in a lock-step way (Single Instruction Multiple Thread).

CUDA Threads and Warps (ii)

- Individual threads (in the warp) start together at the same program address and move forward in lock step.
- A warp executes one common instruction at a time.
- Note that if for some reason (data dependant conditional branch) the threads diverge then the warp serially executes each path taken.
- For eg: if we have 64 threads in a warp and there is a branch were 32 threads execute branch A while the rest execute branch B, the processor while first execute those threads on branch A then those on branch B in sequence (or vice versa).

CUDA Threads and Warps (ii)

- Individual threads (in the warp) start together at the same program address and move forward in lock step.
- A warp executes one common instruction at a time.
- Note that if for some reason (data dependant conditional branch) the threads diverge then the warp serially executes each path taken.
- For eg: if we have 64 threads in a warp and there is a branch were 32 threads execute branch A while the rest execute branch B, the processor while first execute those threads on branch A then those on branch B in sequence (or vice versa).

OpenCL (Khronos Group)

- Open Compute Language is a Khronos group effort into coming up with an open standards parallel computational model.
- Very good industry support ...
- Abstracts all computational resources in a system.
  - CPUs, GPUs
- Programming based on C99
- A group of **devices** are contained in a **host**

OpenCL (Execution Model)

- Kernel
  - Basic unit of executable code
- Program
  - Collection of Kernels and other functions
  - Analogous to a dynamic library
- Applications queue kernel execution instances
  - Queued in order
  - Executed in-order or out-of-order
  - Queues are used to submit work to the devices … It is the CUDA to launching a kernel using the <<< >>> notation.

OpenCL Work (Groups / Items)

- Kernels are executed across a global domain of *work-items* … i.e. one work-item per computation executed in parallel

- Work-items are grouped in local *workgroups* which have shared local memory and synchronisation.

OpenCL Work-Item Identifiers

- Each work-item is aware of what element of the problem it is working on … each work-item can query it's own local and global id.
- Each work-item (and work-group) can be identified within the kernel
- Check next example using get_global_id

Kernel Basic Example (using openCL)

```
__kernel void
Sum(__global const float *a1,
    __global const float *a2,
    __global float *answer)
{
  int xid = get_global_id(0);
  answer[xid] = a1[xid] + a2[xid]
}
```

OpenCL (imp) Functions

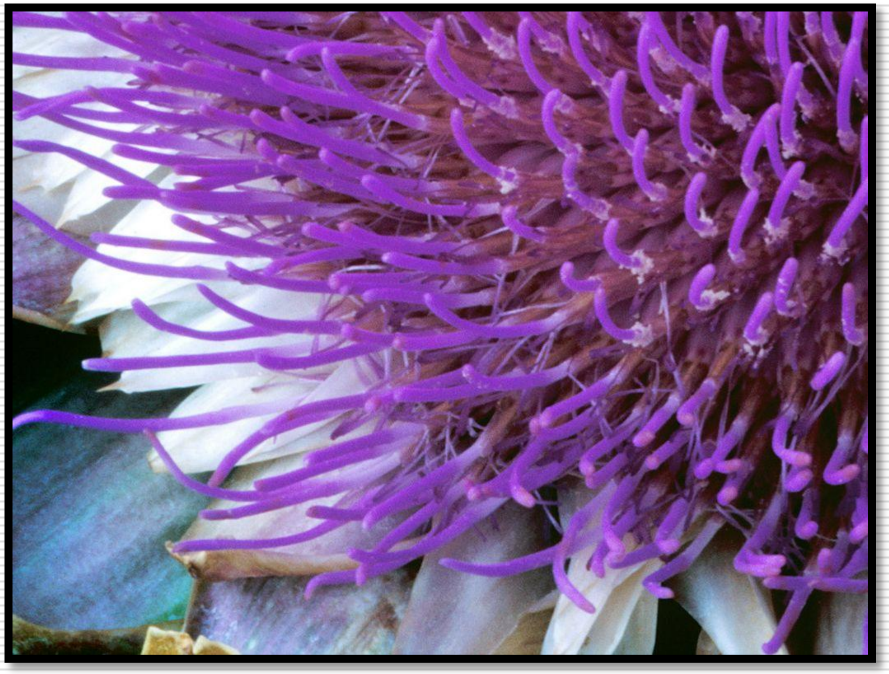
- clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &device, NULL)
- clGetDeviceInfo(device, CL_DEVICE_VENDOR, sizeof(vendor_name), vendor_name, &returned_size)
- clBuildProgram(program[0], 0, NULL, NULL, NULL, NULL)
- mem = clCreateBuffer(context, CL_MEM_READ_ONLY, buffer_size, NULL, NULL)
- clEnqueueWriteBuffer(cmd_queue, mem, CL_TRUE, 0, buffer_size, (void*)a, NULL, NULL);

OpenCL (imp) Functions cont …

- clSetKernelArg(kernel[0], 0, sizeof(cl_mem), &mem);
- clEnqueueNDRangeKernel(cmd_queue, kernel[0], 1, NULL, &global_work_size, NULL, 0, NULL, NULL);
- clEnqueueReadBuffer(cmd_queue, ans_mem, CL_TRUE, 0, buffer_size, results, 0, NULL, NULL)

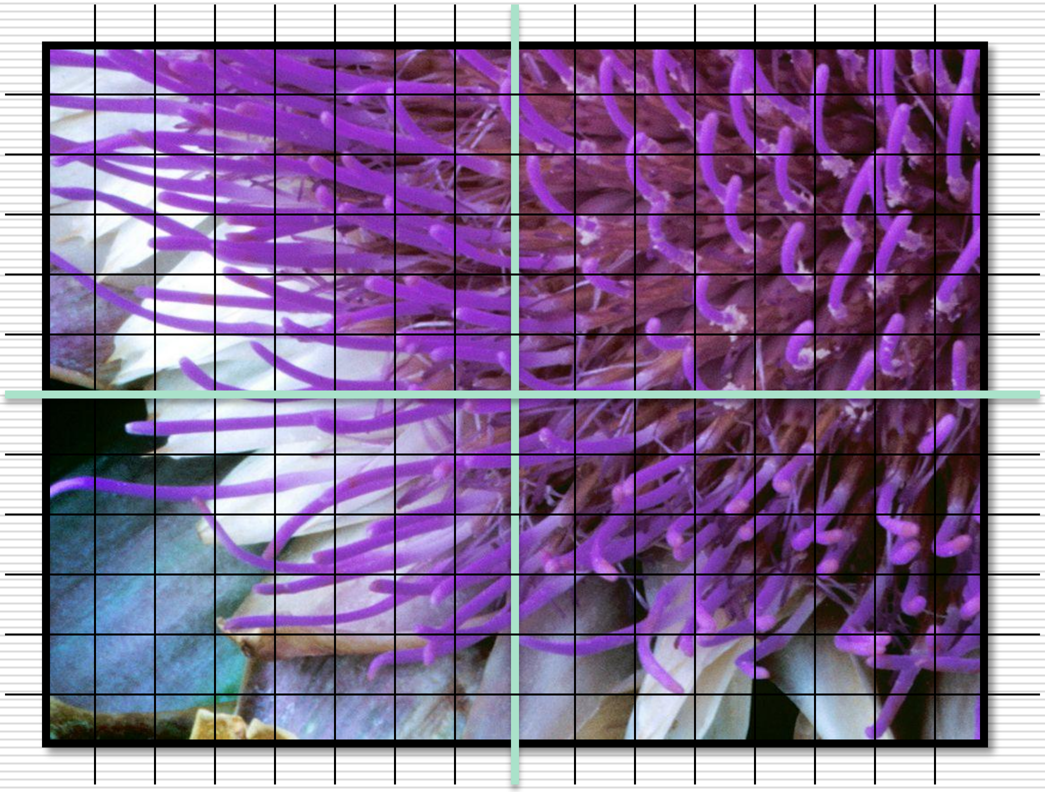
OpenCL example (code start -> end)

- As an example let's assume that we are going to be carrying some calculation (filters) on a 2d image.



OpenCL example (code start -> end)

- Alignment the workgroups and work- items ....



Program Structure *(macresearch.org)*

```
//variable declarations
cl_int err;
cl_context context;
cl_device_id devices;
cl_command_queue cmd_queue;

//Initialisation of structures
err = clGetDeviceIDs(CL_DEVICE_TYPE_GPU, 1, &devices,    NULL);
context = clCreateContext(0, 1, &devices, NULL, NULL, &err);
cmd_queue = clCreateCommandQueue(context, devices, 0,  NULL);
```

Program Structure (macresearch.org)

```
//create the memory buffer ... Set it to READ_ONLY
cl_mem ax_mem = clCreateBuffer(context, CL_MEM_READ_ONLY,
atom_buffer_size, NULL, NULL);

//copy the buffer to the device global memory
err = clEnqueueWriteBuffer(cmd_queue, ax_mem, CL_TRUE, 0,
atom_buffer_size, (void*)ax, 0,NULL,NULL);

//make sure data is updated.
clFinish(cmd_queue);
```

Program Structure *(macresearch.org)*

```
//Load the program from source ...
program[0] = clCreateProgramWithSource(context,1,
(const char**)&program_source, NULL, &err);

//compile the program ...
err = clBuildProgram(program[0], 0, NULL, NULL, NULL, NULL);

//associate the compiled binary
kernel[0] = clCreateKernel(program[0], "mdh", &err);
```

Program Structure *(macresearch.org)*

```
//declare the global and local dimensions
size_t global_work_size[2], local_work_size[2];

//set the dimensions ...
global_work_size[0] = nx; global_work_size[1] = ny;
local_work_size[0] = nx/2; local_work_size[1] = ny/2;
// and the arguments to the kernel ... i.e. The memory locations from where to read
err = clSetKernelArg(kernel[0], 0, sizeof(cl_mem), &ax_mem);
// add to the compute queue ...
err = clEnqueueNDRangeKernel(cmd_queue, kernel[0], 2, NULL,
&global_work_size, &local_work_size,
0, NULL, NULL);
```

Program Structure *(macresearch.org)*

```
//read back the results

err = clEnqueueReadBuffer(cmd_queue, val_mem, CL_TRUE, 0, grid_buffer_size, val, 0, NULL,
        NULL);

//and clean up memory
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cmd_queue);
clReleaseContext(context);
```

N-Body Simulation (i)

- One of your assignments ... Is perfectly suited for deployment on a GPGPU (CUDA/OpenCL/DirectCompute)
- Recall the n-body simulation which simulates the evolution of a system of bodies in which each body interacts and effects every other body.
- In particular the O(N^2) component of the algorithm (near field all-pairs brute force comparison between bodies) really lends itself to parallelisation.
- Let’s take N to be 20,000 bodies ... And assume that 20flops are required to determine the interaction between two bodies. In order to perform a real time simulation we need (20000) (20000) (20) = 8Mn flops / simulation step. This means that if we want RT at 30FPS then we need to perform 8Mn * 30 = 240Giga flops / second

N-Body Simulation (ii)

- Recall the formula used to calculate the gravitational attractive potential between two bodies **i** and **j** :

$$\boldsymbol{f}_i = G\frac{m_i m_j}{\|\boldsymbol{r}_{ij}\|^2} \cdot \frac{\boldsymbol{r}_{ij}}{\|\boldsymbol{r}_{ij}\|}$$

- Where f_{ij} represents the force vector on body i caused by gravitational attraction to body j,
- m_i and m_j represent the masses of i and j respectively,
- r_{ij} represents the distance vector between bodies i and j,
- G is the gravitational constant
- Note that f and r are both vectors. The right part of the equation gives the direction (unit vector) of the force.

N-Body Simulation (iii)

- Hence, given this formula one can calculate the total force (F_i) applied on a body *i* by the rest of the N-1 bodies …
- Which is a summation of vectors:

$$F_i = \sum_{\substack{i \leq j \leq N, \\ j \neq i}} \boldsymbol{f}_{ij} = G m_i \frac{m_j \boldsymbol{r}_{ij}}{\left\| \boldsymbol{r}_{ij} \right\|^3}$$

- In order to avoid collisions between bodies a softening factor ε in the denominator is added. (details not shown above).
- Recall also that to integrate over time we use $F_i = m_i\ a_i$ in order to calculate the acceleration which will give us the new position of i.

N-Body Simulation (iv)

- Rearranging the formula to calculate acceleration *a* gives us :

$$a_i \approx G. \sum_{i \leq j \leq N} \frac{m_j \, \boldsymbol{r}_{ij}}{\left(\left\| \boldsymbol{r}_{ij} \right\|^2 + \varepsilon^2 \right)^{\frac{3}{2}}}$$

- In the next slide we shall see how to implement the computation of a_{ij} which gives us an update in acceleration given the interaction between bodies i and j .... For any body i ... Acceleration a is updat N-1 times. G is multiplied at a later stage (not in the kernel code)

Code (from Paper in GPU Gems 3)

```
__device__ float3  bodyBodyInteraction(float4 bi, float4 bj, float3 ai)   {
    float3 r;
    // r_ij  [3 FLOPS]
    r.x = bj.x - bi.x;   r.y = bj.y - bi.y;   r.z = bj.z - bi.z;
    // distSqr = dot(r_ij, r_ij) + EPS^2  [6 FLOPS]
    float distSqr = r.x * r.x + r.y * r.y + r.z * r.z + EPS2;
    // invDistCube =1/distSqr^(3/2)  [4 FLOPS (2 mul, 1 sqrt, 1 inv)]
    float distSixth = distSqr * distSqr * distSqr;
    float invDistCube = 1.0f/sqrtf(distSixth);
    // s = m_j * invDistCube [1 FLOP]
    float s = bj.w * invDistCube;
    // a_i =  a_i + s * r_ij [6 FLOPS]
    ai.x += r.x * s;   ai.y += r.y * s;  ai.z += r.z * s;
    return ai;

}
```

N-Body Simulation (strategy)

- We can use a grid of NxN locations to store all pair-wise forces. The total **F** for **i** can then be calculated by summing the elements in their respective row i.

- This approach requires $O(N^2)$ memory. Possible but performance would be hindered by memory bandwidth (between global and local)!!

- In order to achieve some memory reuse (in the block) we need to serialise some of the computation.

- To increase the reuse of data (in the shared memory), one approach is to arrange computation in such a way as to evaluate each row sequentially (updating the acceleration), and separate rows are evaluated in parallel.

N-Body Simulation (strategy -ii)

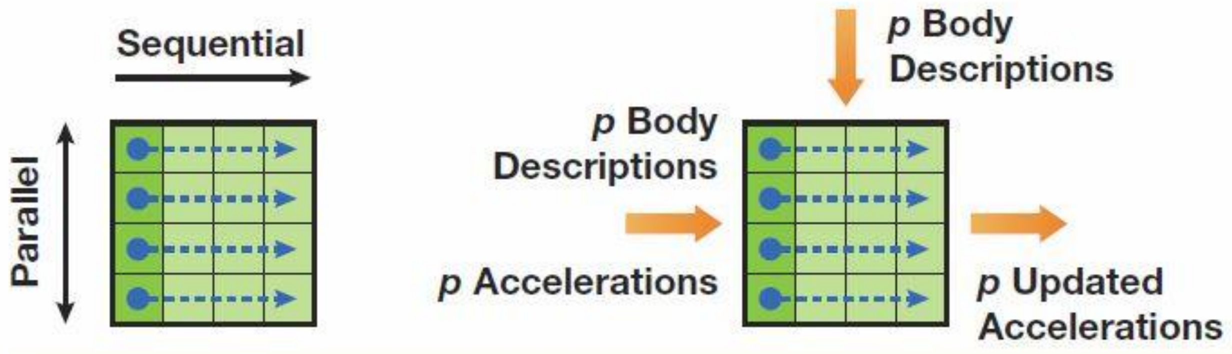


Figure 31-2. A Schematic Figure of a Computational Tile
Left: Evaluation order. Right: Inputs needed and outputs produced for the p^2 interactions calculated in the tile.

Source: Chapter 31 GPU Gems 3 – Nvidia

- A tile is evaluated by *p* threads performing the same sequence of operations (using different data)
- Each thread updates the acceleration of one body as a result of its interactions with p other bodies

N-Body Simulation (strategy –iii)

```
  __device__ float3  tile_calculation(float4 myPosition, float3 accel)

{
    int i;
    extern __shared__ float4[] shPosition;
    for (i = 0; i < blockDim.x; i++) {
        accel = bodyBodyInteraction(myPosition, shPosition[i], accel);
    }
    return accel;
}
```

- This code will compute the accelerations for a tile (pxp). Note the for loop which is essentially forcing a sequence on the computation.

CUDA kernel (ii)

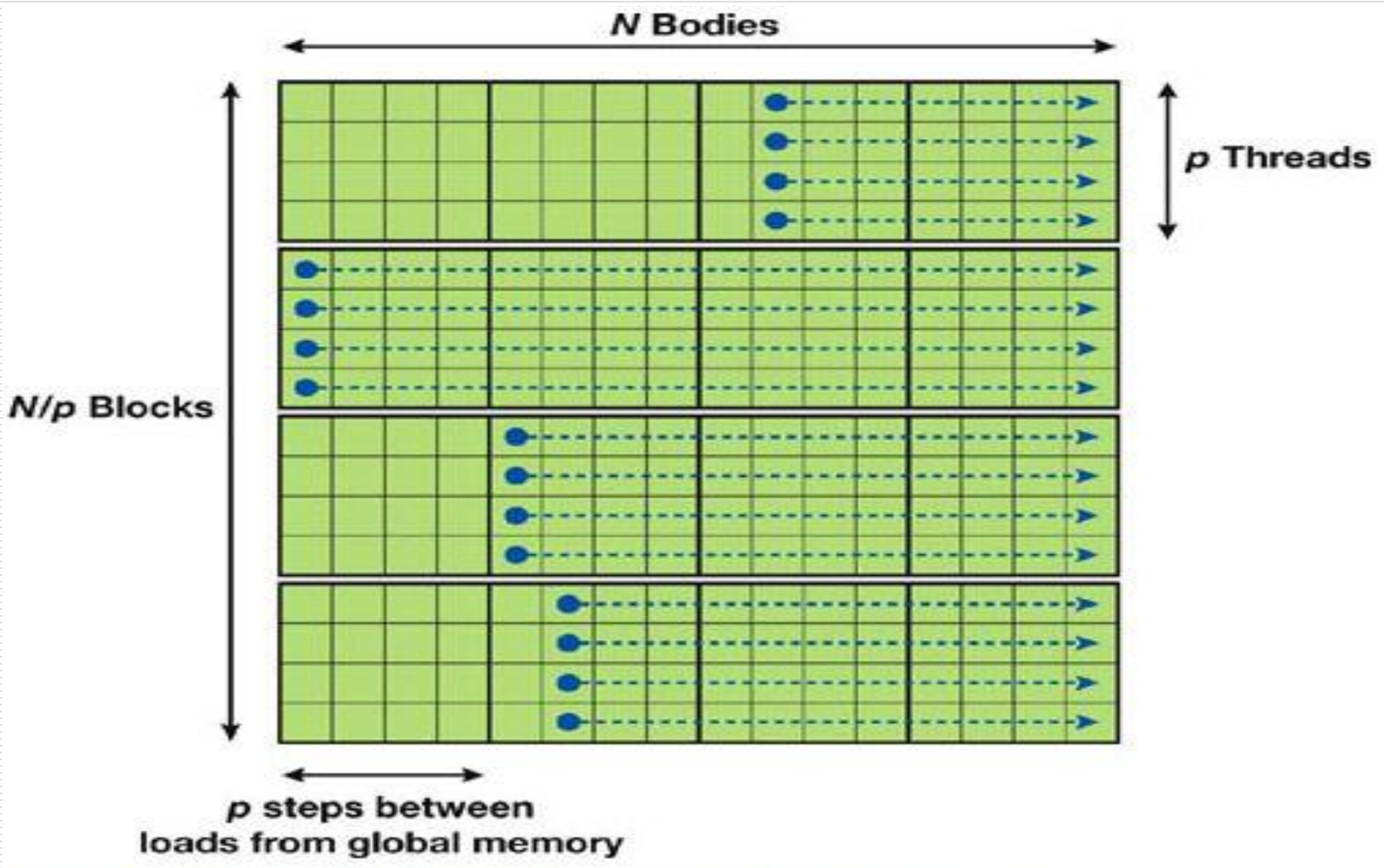


Figure 31-4 The Grid of Thread Blocks That Calculates All Forces

CUDA kernel (i)

```
  __global__ void
calculate_forces(void *devX, void *devA)
{
 extern __shared__ float4[] shPosition;
 float4 *globalX = (float4 *)devX;
 float4 *globalA = (float4 *)devA;
 float4 myPosition;
 int i, tile;
 float3 acc = {0.0f, 0.0f, 0.0f};
 int gtid = blockIdx.x * blockDim.x + threadIdx.x;
 myPosition = globalX[gtid];
```

(.... Continued on next slide)

CUDA kernel (ii)

(...continued from previous slide)

```
  for (i = 0, tile = 0; i < N; i += p, tile++) {
    int idx = tile * blockDim.x + threadIdx.x;
    shPosition[threadIdx.x] = globalX[idx];
    __syncthreads();
    acc = tile_calculation(myPosition, acc);
    __syncthreads();
  }
  // Save the result in global memory for the integration step.
   float4 acc4 = {acc.x, acc.y, acc.z, 0.0f};
  globalA[gtid] = acc4;

}
```

Conclusions

- Parallelisation and GPGPU will over the next couple of years take a central role in computing.
- The aim of these lectures was to inform you of what GPGPU is all about and perhaps get you interested in the area.
- There are many applications that will benfit from the GPGPU model of computation … Mostly of a scientific nature but not just that.