KFUPM HPC Workshop

April 29-30 2015

Mohamed Mekias

HPC Solutions Consultant

Introduction to CUDA programming

Agenda

- **GPU Architecture Overview**
- **Tools of the Trade**
- **Introduction to CUDA C**
- **Patterns of Parallel Computing**
- **Thread Cooperation and Synchronization**
- **The Many Types of Memory**
- **Atomic Operations**
- **Events and Streams**
- **CUDA in Advanced Scenarios**

A reminder of how it all started

- There were no GPUs at the beginning
- Graphic rendering was handled by the CPU
- Then PCI based GPUs were introduced to just handle the graphics
  - Relieved the CPU and allowed the division of labor
  - The CPU computes and the GPU renders
- Then games came in, and the need to program at the pixels level was needed
  - Shaders came in to the picture
    - OpenGL, DirectX are example of shader programming
    - C-like programming languages

Shaders

- Shaders are small programs that run on the GPU
  - Basic operations are
    - Calculate the location of a vertex
    - Calculate the color of a pixel (components)
- Shader languages include:
  - High Level Shader Language (MS DirectX)
  - GLSL, OpenGL
  - C-like languages
  - Intended for graphics (pixel manipulation)
- What if we convert data to texture?
  - Then we run it through a shader
  - Then convert texture back to data
- This is exactly what happened in early 90's
- Rebellion against CPU started

Why GPGPU?

- General-Purpose Computation on GPUs
- Highly parallel architecture
  - Lots of concurrent threads of execution (SIMT)
  - Higher throughput compared to CPUs
    - Even taking into account many cores, hypethreading, SIMD
  - Thus more FLOPS (floating-point operations per second)
- Commodity hardware
  - Commonly available (mainly used by gamers)
  - Relatively cheap compared to custom solutions (e.g., FPGAs)
- Sensible programming model
  - Manufacturers realized GPGPU market potential
  - Graphics made optional
  - NVIDIA offers dedicated GPU platform “Tesla”
    - No output connections

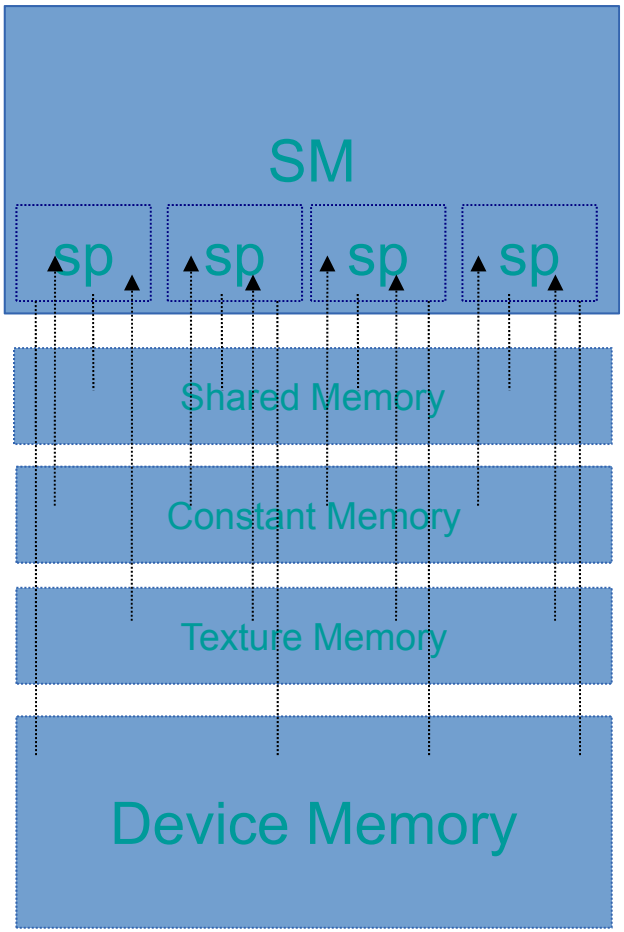
GPGPU programming Frameworks

- Compute Unified Driver Architecture (CUDA)
  - Developed by NVIDIA
  - Extension to C (then C++)
  - Wrappers for other languages and scripting
    - Fortran, Python, MATLAB, MATHEMACTICA etc...
- OpenCL and OpenACC
  - Supported by NVIDIA, CRAY, Intel(OpenCL), AMD(ATI), PGI(now part of NVIDIA)
- AMP (Accelerated Massive Programming)
  - Driven by Microsoft
  - C++ extension and part of latest version of MSVC++
  - Support both NVIDIA and ATI devices
- Other farmeworks
  - Alea, Aparapi, Brook & many more...

Graphic Processor terminology(NVIDIA)

- Streaming Multiprocessor (SM)
  - Has many CUDA cores
  - Each card can have more than one SM
- CUDA core a.k.a
  - Streaming processor
  - Shader unit
- Compute capability
  - Tied to number of cores per SM
  - The higher the better
- Many different types of memory
  - Different access
  - Different performance characteristics

High level architecture (can be more than one SM)



Compute capability

- A number indicating what the card can do
  - Higher is better
  - Checkout http://en.wikipedia.org/wiki/CUDA
  - Current range: 1.0 (early Ge(x) GPUs),...,5.2(Maxwell)
  - KFUPM GPUs Tesla K40 compute capability 3.5
- Why does it matter?
  - Affects both hardware and API support
  - Number of Cuda cores per SM
  - Maximum number of registers per SM
  - Maximum number of instruction per cuda kernel
  - Support for double precision (earlier GPUs had only single precision)
  - Etc...

Programming Tools & libraries

- CUDA-Toolkit
  - Compiler nvcc
  - Libraries
    - cufft, cublas, curand, ...
    - boost++
  - Cuda samples
  - NSIGHT
    - Eclipse template for cuda programming
    - integrates(in other IDEs like MSVC++)
- OpenACC (PGI compiler)
- OpenCL

Let's do the first example: Adding 2 vectors

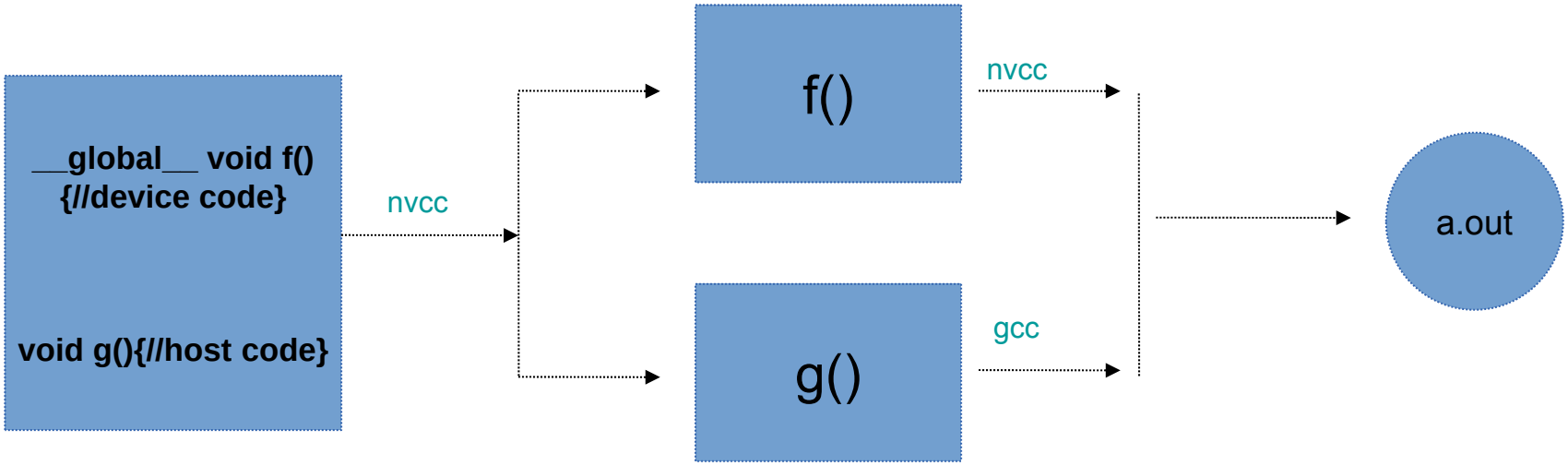
```
#include <stdio.h>
#include <cuda_runtime.h>
__global__ void vectorAdd(const float *A, const float *B, float *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
        C[i] = A[i] + B[i];
}
int main(void)
{
    int numElements = 32768;
    size_t size = numElements * sizeof(float);
    float *h_A, *h_B, *h_C;
    h_A= (float *)malloc(size); h_B = (float *)malloc(size); h_C = (float *)malloc(size);
    for (int i = 0; i < numElements; ++i) //initialize
    {
        h_A[i] = rand()/(float)RAND_MAX;
        h_B[i] = rand()/(float)RAND_MAX;
    }
    float *d_A, *d_B, *d_C;
    cudaMalloc((void **)&d_A, size); cudaMalloc((void **)&d_B, size);cudaMalloc((void **)&d_C, size);
```

Approximating pi

$$4\int_0^1 dx/(1+x^2)$$

More on nvcc

- Not really a compiler
  - Relies on host C/C++ compilers like gcc,msvc++
  - Splits code into host code and device code
    - Hands over host code to native compiler
    - Generate PTX code for device parts
    - Cuda Driver converts PTX to binary code



What is PTX?

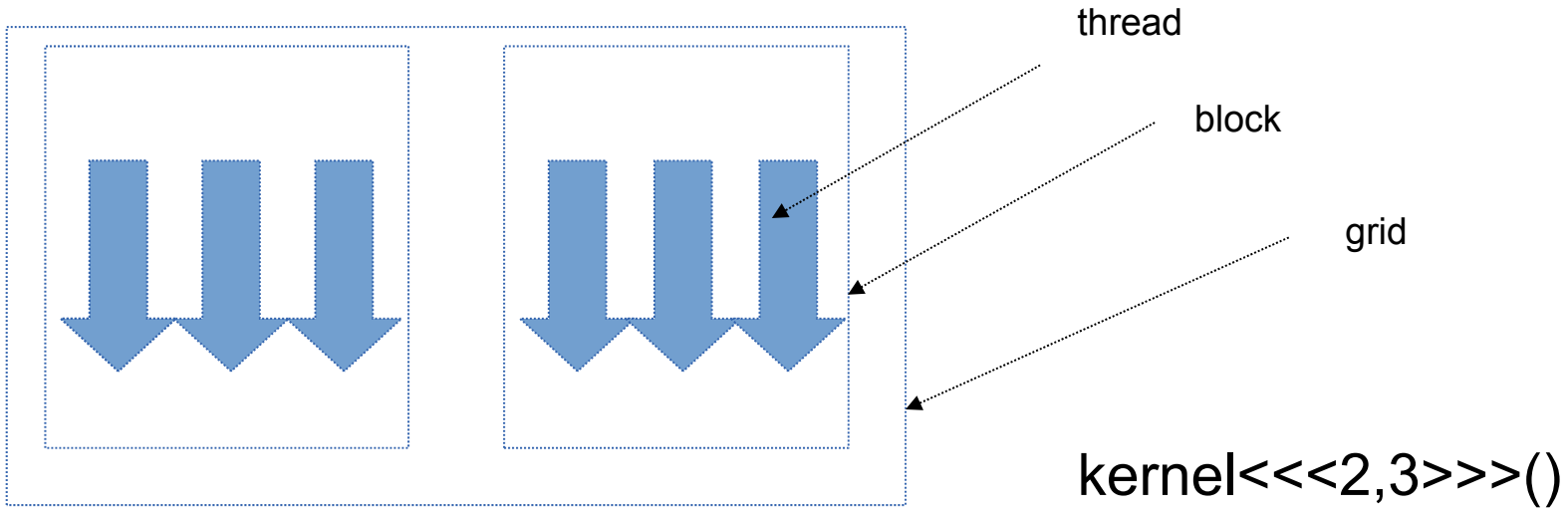
- PTX is the ‘assembly language’ of CUDA
  - Similar to .NET IL or Java bytecode
  - Low-level GPU instructions
  - Can be generated by nvcc to see (ugly)
  - Useful for compiler writers
  - Can be modified and inlined again very much like inline assembly in C

CUDA Qualifiers

- __global__
  - Defines a kernel
  - Runs on the GPU
  - Called from CPU
  - Takes arguments dim3 (more on this later)
- __device__
  - Runs on the GPU and called from the GPU
  - Can be functions and variables
- __host__
  - Runs on CPU and called from CPU
- Qualifers can be combined
  - __host__ __device__
  - Useful for debugging and portability

Execution Model

- Thread blocks are scheduled to run on available SMs
- Each SM executes one block at a time
- A Thread block is divided into warps
  - Number threads per warp depends on compute capability of the device
- All warps are handled in parallel

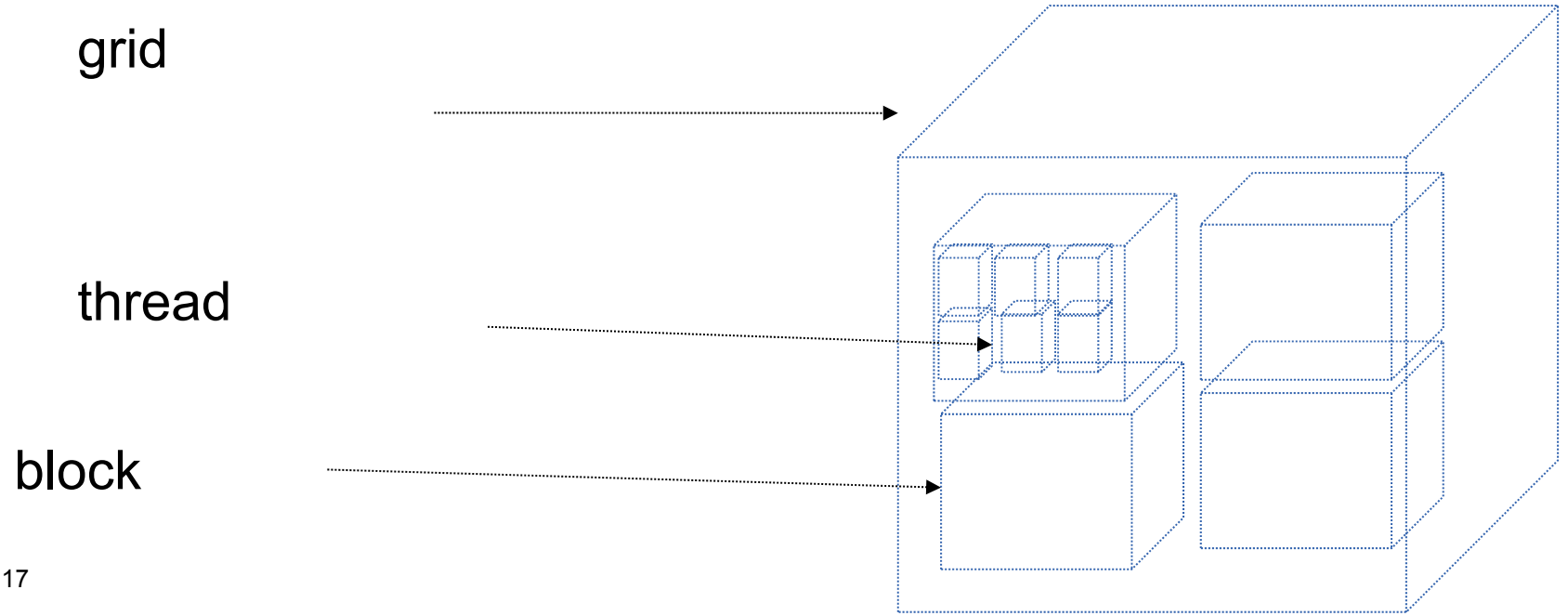


kernel<<<2,3>>>()

Run the function kernel on 2 blocks with 3 threads

Dimensions

- Expressed thread execution by triple bracket
  - <<<*n*,*m*>>>, where *n* is the number of blocks and *m* is the number of threads per block
  - That is a grid of *n* blocks of *m* threads each
- In reality execution is expressed with a 3d object
  - A 3-d grid of 3-d blocks (6d total)



Dimensions (cont...)

- Can have 1d and 2d by setting to 1 the other dimensions
- dim3 structure is a simple container with x,y,z values
- Automatic conversion of
  - <<<a,b>>> -------> block (a,1,1) by threads (b,1,1)
- Thread variables
  - blockIdx, gives location in the grid
    - Which block of the grid are we in?
  - gridDim, size of the grid
  - threadIdx, current thread in the thread block
  - blockDim, size of the thread block

Dimension Limitations

- Block Sizes:
  - MAX_BLOCK_DIMX (512)
  - MAX_BLOCK_DIMY (512)
  - MAX_BLOCK_DIMZ (64)
- Grid sizes:
  - MAX_GRID_DIMX (65535)
  - MAX_GRID_DIMY (65535)
  - MAX_GRID_DIMZ (1)
- Number of threads
  - MAX_THREADS_PER_BLOCK (512)
  - MAX_THREADS_PER_MULTIPROCESSOR (1024)

Some rules you need to keep in mind of data access

- Data in CUDA is in arrays of basic data types
  - No data structures
  - No compiler auto-parallelization
  - No parallel data structures
  - No CPU SIMD-like instructions ====> No instruction pipelining
  - Very basic multiprocessor
- Beware when writing in C++
  - No support of c++11
- Thread indexing can be very confusing
  - Up to 6d (3d grid x 3d thread block)
- Most applications have typical 1d input
  - 2d possible but 3d are rare
- Some examples of thread indexing
  - 1 block N threads --->threadIdx
  - 1 block, MxN threads ---> threadIdx.y * blockDim.x + threadIdx.x
  - N blocks, M threads ---> blockIdx.x * gridDim.x + threadIdx.x

Error handling

- CUDA doesn't throw exceptions
  - Failures are silent
- But has some basic error handling functionality
  - Core functions return cudaError_t
  - Check returned object against cudaSuccess
  - In case of failure call
    - cudaGetErrorString()
- Beware that libraries may define their own error objects
  - curand has curandStatus_t