CUDA Programming (NVIDIA)

**Compute Unified Device Architecture:**

- C programming on GPUs
- Requires no knowledge of graphics APIs or GPU programming
- Access to naive instructions and memory
- Easy to get started
- Stable, free, documented, supported
- Windows and Linux
GPU

GPU is viewed as a compute device operating as a coprocessor to the main CPU (host)

Origin: Operators on images (filter, rendering,..)

- For data parallel, cost intensive functions or functions that are executed many times, but on different data (loops)
- A function compiled for the device is called a kernel
- The kernel is executed on the device as many different threads
- Host (CPU) and device (GPU) manage their own memory
- Data can be copied between them (slow)
3rd Level of Parallelism

MPI

OpenMP
The computational grid consists of a grid of thread blocks.

Each thread executes the kernel.

The application specifies the grid and the block dimensions.

Grid layouts can be 1D, 2D, 3D.

Maximal sizes are determined by GPU memory and kernel complexity.

Each block has unique block ID.

Each thread has unique thread ID (within its block).
**Example: Matrix Addition**

**CPU Program**

```c
void add_matrix
   ( float* a, float* b, float* c, int N ) {
   int index;
   for ( int i = 0; i < N; ++i )
      for ( int j = 0; j < N; ++j ) {
         index = i + j*N;
         c[index] = a[index] + b[index];
      }
   }

int main() {
   add_matrix( a, b, c, N );
}
```

**CUDA Program**

```c
__global__ add_matrix
   ( float* a, float* b, float* c, int N ) {
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;
   int index = i + j*N;
   if ( i < N && j < N )
      c[index] = a[index] + b[index];
   }

int main() {
   dim3 dimBlock( blocksize, blocksize );
   dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
   add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N );
}
```

The nested for loops are replaced with an implicit grid.
CUDA exposes all the different types of memory on the GPU
## Memory Model II

<table>
<thead>
<tr>
<th>Overview</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Per thread</td>
<td>Read-Write</td>
<td></td>
</tr>
<tr>
<td>Local memory</td>
<td>Per thread</td>
<td>Read-Write</td>
<td></td>
</tr>
<tr>
<td>Shared memory</td>
<td>Per block</td>
<td>Read-Write</td>
<td>For sharing data within a block</td>
</tr>
<tr>
<td>Global memory</td>
<td>Per grid</td>
<td>Read-Write</td>
<td>Not cached</td>
</tr>
<tr>
<td>Constant memory</td>
<td>Per grid</td>
<td>Read-only</td>
<td>Cached</td>
</tr>
<tr>
<td>Texture memory</td>
<td>Per grid</td>
<td>Read-only</td>
<td>Spatially cached</td>
</tr>
</tbody>
</table>

Explicit GPU memory allocation and deallocation via `cudaMalloc()` and `cudaFree()`

Access to GPU memory via pointers

Copying between CPU and GPU memory:
A slow operation: minimize this!
Multiple levels of parallelism:

- Thread block:
  Up to 512 threads per block
  Communicate via shared memory
  Threads guaranteed to be resident
  threadIdx, blockIdx
  __syncthreads()

- Grid of thread blocks:
  f<<<N, T>>>( a, b, c)
  Communication via global memory

  f: function, kernel
  N,T: size of threads
  (a,b,c): data
CUDA Programming Language

- Minimal C extensions
- A runtime library
  - A host (CPU) component to control and access GPU(s)
  - A device component
  - A common component:
    Built-in vector types, C standard library subset
- Function type qualifiers
  - Specify where to call and execute a function
    __device__, __global__, __host__
- Variable type qualifiers
  __device__, __constant__, __shared__
- Kernel execution directive
  foo<<<GridDim, BlockDim>>>(…)
- Built-in variables for grid/block size and block/thread indices
CUDA Compiler

Source files must be compiled with the CUDA compiler nvcc

CUDA kernels are stored in files ending with .cu

NVCC uses the host compiler to compile CPU code
NVCC automatically handles #include‘s and linking

Write kernels+CPU caller in .cu files
Compile with nvcc
Store signature of CPU called in header file
#include header file in C(++) sources
GPU Runtime Component

Only available on the GPU:

- Less accurate, faster math functions \( \_\_\sin(x) \)
- `syncthreads()`: wait until all threads in the block have reached this point
- Type conversion functions, with rounding mode
- Type casting functions
- Texture functions
- Atomic functions:
  - Guarantees that operation (like add) is performed on a variable without interference from other threads
CPU Runtime Component

Only available on the CPU:

- Device management:
  Get device properties, multi-GPU control, …
- Memory management:
  cudaMalloc(), cudaMemcpy(), cudaFree(),
- Texture management
- Asynchronous concurrent execution
- Built-in vector types: i.e. float1, float2, int3, ushort4, …
- Mathematical functions: standard math.h on CPU,
const int N = 1024; const int blocksize = 16;

__global__ void add_matrix( float* a, float *b, float *c, int N )
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N;
    if ( i < N && j < N )
        c[index] = a[index] + b[index];
}

int main() {
    float *a = new float[N*N]; float *b = new float[N*N]; float *c = new float[N*N];
    for ( int i = 0; i < N*N; ++i ) {
        a[i] = 1.0f; b[i] = 3.5f;
    }
    float *ad, *bd, *cd;
    const int size = N*N*sizeof(float); cudaMalloc( (void**)&ad, size );
    cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
    cudaMalloc( (void**)&bd, size );
    cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );
    dim3 dimBlock( blocksize, blocksize );
    dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
    add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );
    cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );
    cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
    delete[] a; delete[] b; delete[] c;
    return EXIT_SUCCESS;
}
Execution Model

A GPU consists of $N$ multiprocessors (MP)
Each MP has $M$ scalar processors (SP)
Each MP processes batches of blocks
   A block is processed by only one MP
Each block is split into SIMD groups of threads called warps
   A warp is executed physically in parallel
A scheduler switches between warps
A warp contains threads of consecutive, increasing thread ID
The warp size is 32 threads today
Costs of Operations

4 clock cycles: Floating point: add, multiply, fused mult.
   - add
   Integer add, bitwise op., compare, min, max

16 clock cycles: reciprocal, square root, __log, 32-bit int mult

32 clock cycles: __sin(x) , __cos(x) , __exp(x)

36 clock cycles: Floating point division

Particularly expensive: Integer division, modulo

Double precision will perform at half the speed
Right kind of memory

Constant memory:
  Quite small, \( \approx 20K \)
  As fast as register access if all threads in a warp access the same location

Texture memory:
  Spatially cached
  Optimized for 2D locality
  Neighbouring threads should read neighbouring addresses
  No need to think about coalescing

Constraint: These memories can only be updated from CPU
Accessing global memory

4 cycles to issue on memory fetch but 400-600 cycles of latency!

Likely to be a performance bottleneck

Order of magnitude speedups possible:
Coalesce memory access

Use shared memory to reorder non-coalesced addressing
Coalescing (Assembling)

For best performance, global memory access should be coalesced

A memory access coordinated within a warp
A contiguous, aligned region of global memory
- 128 bytes – each thread reads a float or int
- 256 bytes – each thread reads a float2 or int2
- 512 bytes – each thread reads a float4 or int4
- float3s are not aligned

Warp base address (WBA) must be a multiple of 16*sizeof(type)

The k-th thread should access the element at WBA+k

These restrictions apply to both reading and writing
Coalesced memory access

Coalesced memory access:
Thread $k$ accesses $WBA + k$
Coalesced memory access:
Thread $k$ accesses $WBA + k$

*Not all threads have to participate*
Coalesced memory access

Non-coalesced memory access:

*Misaligned starting address*
Coalesced memory access

Non-coalesced memory access:
Non-sequential access
Coalesced memory access

Non-coalesced memory access:
Wrong size of type
Software for GPU

CUDA

OpenCL: Vendor-independent industrial standard

DirectCompute: GPU computing from Microsoft

...
OpenCL

C-based language for GPU kernels + device kernels

Plus low-level device API (application programming interface)
- Same flavor as CUDA
- JIT (just in time) compilation of kernel programs
- Portable - but inevitable optimization required for every platform

Managed by Khronos group (non-profit organization)
- All major vendors participate
- This is the cross-vendor industry-standard
Work Items – N-D Range

OpenCL execution model:
- define an N-d computation domain
- execute kernel for each point in the domain

Global dimensions: 1024 x 1024
Local dimensions: 128 x 128
work group – execute together

Synchronization between work-items possible only within workgroups using barriers and memory fences

Choose the dimensions That are the „best“ for your algorithm

Cannot synchronize outside of a workgroup
Parallelizing For Loop

Scalar:

```c
void scalar_mul(const float *a, const float *b, float *c, int n)
{
    int i;
    for (i=0; i<n; i++)
        c[i] = a[i] * b[i];
}
```

Data Parallel:

```c
kernel void dp_mul(global const float *a, global const float *b, global float *c)
{
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
} // execute over "n" work-items
```

Kernel executed n times, once for each work item

Get the index of the work item
OpenCL Objects

Setup:
- Devices (CPU, GPU, Cell,..)
- Contexts (collection of devices)
- Queues (submit work to the device)

Memory:
- Buffers (blocks of memory, kernels can acces however they like)
- Images (2D or 3D formatted images)

Execution:
- Programs (collections of kernels)
- Kernels

Synchronisation:
- Events
DirectCompute

GPGPU under Windows

- Microsoft API / Windows standard for all GPU vendors
- General-purpose GPU computing under Windows
- released with DirectX11 / Windows 7
- Supports all CUDA-enabled devices and ATI GPUs
- Low-level API for device management and launching of kernels
- Defines HLSL-based language for compute shaders (High Level Shading Language)
Application-level integration

Matlab:
   Parallel Computing Toolbox,
   Jacket / AccelerEyes. GPU acceleration engine

Mathematica
LabView
PetSc
Trilinos
OpenFoam, Ansys,..
Application-specific Libraries

CUsparse: NVIDIA library for sparse matrix vector operations

CUBLAS: NVIDIA library for dense linear algebra

CUFFT: NVIDIA library for Fast Fourier Transforms

CUSP: NVIDIA library for sparse linear algebra

Thrust: A template library for CUDA applications (sort, reduce,..)

MAGMA: LAPACK for HPC on heterogeneous architectures

OpenCurrent: PDE, Gauss-Seidel, multigrid
Application-specific Libraries

ViennaCL: Scientific computing library, C++, OpenCL
BLAS, iterative methods, preconditioners

LAMA: Library for Accelerated math Applications
BLAS, various sparse matrix storage formats

FEAST: Finite Element Analysis and Solution Tools

PARDISO: Fast direct solver for sparse problems
Using BLAS functionality of GPUs