Parallel computing

Parallel computing is not new but:

- often it is now massively parallel
- we cannot wait until our old serial program runs faster (it never will)
- hence today programming is parallel programming
- we are probably restrained in first thinking about an algorithm in a serial way (implicit serialism in programming languages) - example

Many tools are tried for parallel computing:

- very explicit ones like MPI (for cluster machines and supercomputers)
- lightweight language extensions such as OpenMP, OpenACC, Array Building Blocks (ArBB), Cilk Plus, ...
- domain-specific languages: CUDA, OpenCL, OpenGL, ...
- intelligent compilers, automatic parallelizers: PGI Compilers

GPU computing

traditional interpretation of GPU computing
traditional interpretation of GPU computing

- Core i7 IvyBridge i7-3870: ≈ 21 GB/s
- NVIDIA Tesla K10: 4580 GFLOP/s (single precision)

GPU computing

- GPU computing

GPU basics

08/10/2018 4 / 36
Latency vs. throughput

CPU vs. GPU hardware

- CPU optimized for low-latency access to cached data
- extensive logic for branch prediction and out-of-order execution
- do an unpredictable scalar job as fast as possible

GPU optimized for data-parallel throughput computations
- latency hiding
- do as many simple, deterministic jobs in parallel as possible

Latency hiding

- CPU must minimize latency of individual thread for responsiveness
- GPU hides latency through interleaved execution

General processing flow
Copy data from host memory to GPU main ("global") memory.
3. Load GPU code, cache data on GPU for performance.
4. Data is persistent in GPU memory, continuous exchange is possible.
5. Copy result from GPU to CPU memory.
GPU architecture

GPGPU history

NVIDIA
- introduced CUDA in 2007
- developed into a fully blown ecosystem
- series of computing cards
- academic support programs
  - CUDA professorships
  - CUDA research centers
  - CUDA teaching centers

ATI
- ATI Stream introduced in 2007
- less viral marketing
- higher peak performance
- somewhat less flexible architecture

Other architectures
- Intel MIC
  - started as Larabee in 2006
  - prototype board Knight’s Ferry (2010)
    - 32 cores with 4 threads/core
    - 2 GB DDR5 memory
  - Knight’s Corner, released in 2012
    - more than 50 cores per chip
  - Knight’s Landing, released in 2016
    - up to 72 cores per board
    - used in a number of recent supercomputers
- (some) special-purpose machines
  - ANTON by D. E. Shaw research, composed of purpose-built ASICs
  - Janus, FPGA based machine for spin models
  - QPace, based on the Cell processor known from Playstation 3
  - GRAPE, based on FPGAs and used for astrophysical N-body simulations

GPU architecture

GPU architecture: main components

Main memory
- up to 16GB in current GPUs
- maximum bandwidth up to 900 GB/s on Volta (V100)
- accessible from CPU and GPU sides
- large latency (see below)
- optional error correction (ECC on/off, Fermi onwards)

Several multiprocessors
- similar to a multi-core CPU
- each has its own set of registers, scheduler, caches etc.
  + scheduling units, PCIe logic etc.

GPU architecture: streaming multiprocessor (Fermi)

SM components
- 32 cores per SM (Fermi)
  - 32 fp32 ops/s
  - 32 int32 ops/s
- Maxwell: 192 core, Pascal: 128
- 2/4 warp schedulers (warp=32 threads)
  - max. 1536/2048 resident threads
- extra special function units (4 for Fermi)
- 64/96 KB cache on die, re-configurable as 16 KB cache + 32 KB shared memory or vice versa
- 32–128K 32-bit registers

NVIDIA microarchitectures

Latest generations: Volta (2017/18), Turing (2019/20)
Disclaimer: this is promotional material!
GPU architecture: computing core

Computing core
- Integer and floating-point units:
  - IEEE-2008 compliant floating point arithmetic (starting from Fermi)
  - Fused multiply-and-add instruction in hardware
- Logic unit
- Move, compare unit
- Branch unit

GPU architecture

Definitions

Kernel GPU program that runs on a grid of threads

Thread scalar execution unit

Warp block of 32 threads executed in lockstep

Block a set of warps executed on the same SM

Grid a set of blocks usually executed on different SMs

GPU computation frameworks

GPGPU = General Purpose Computation on Graphics Processing Unit

“Old” times: use original graphics primitives
- OpenGL
- DirectX

Vendor specific APIs for GPGPU:
- NVIDIA CUDA: library of functions performing computations on GPU (C, C++, Fortran), additional preprocessor with language extensions
- ATI/AMD Stream: similar functionality for ATI GPUs

Device independent schemes:
- superseded approaches: BrookGPU, AMD Stream, Sh
- OpenCL (Open Computing Language): open framework for parallel programming across a wide range of devices, ranging from CPUs, Cell processors and GPUs to handheld devices
- OpenACC: pragma based parallelization of code parts

NVIDIA GeForce GTX Titan

- Kepler generation card
- 2688 streaming processor cores
- 837 MHz clock frequency (1.5 GHz memory clock)
- single precision peak performance 4.4 TFLOP/s
- 1.5 GB GDDR5 RAM
- memory bandwidth 288 GB/s
Thread hierarchy

- Parallel portions of an application are executed on GPU as kernels.
  - one kernel is executed at a time (later on modified in concept of streams)
  - each kernel executes in many threads, but on one device
- Compare CUDA threads to CPU threads
  - CUDA threads are lightweight
    - very little creation overhead
    - low cost of thread switching
  - CUDA needs thousands of threads for efficiency
- Kernels are executed by an array of threads
  - all threads run the same code
  - each thread has a unique threadid for control decisions and memory access
- Kernel launched are in grid of thread blocks
  - threads within block cooperate via shared memory
  - threads within a block can synchronize
  - threads within different blocks cannot cooperate (or only via global memory)
  - block executes on one SM and does not migrate
- allows programs to transparently scale to GPUs with different numbers of cores

CUDA code: C with some extra reserved words

Consider a simple “SAXPY” computation, i.e., “Single-Precision \(A \cdot X + Y\)”.  

Standard C code

```c
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);
```

CUDA C code

```c
__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);
```

CUDA variables

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
<th>Penalty/Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>int var;</td>
<td>register</td>
<td>thread</td>
<td>thread</td>
<td>1X</td>
</tr>
<tr>
<td>int array_var[10];</td>
<td>local</td>
<td>thread</td>
<td>thread</td>
<td>100X (pre-Fermi)</td>
</tr>
<tr>
<td><strong>shared</strong> int shared_var;</td>
<td>shared</td>
<td>block</td>
<td>block</td>
<td>10X</td>
</tr>
<tr>
<td><strong>device</strong> int global_var;</td>
<td>global</td>
<td>grid</td>
<td>application</td>
<td>100X</td>
</tr>
<tr>
<td><strong>constant</strong> int constant_var;</td>
<td>constant</td>
<td>grid</td>
<td>application</td>
<td>1X</td>
</tr>
</tbody>
</table>

- automatic scalar variables reside in registers, compiler will spill into local memory in shortage of registers (use `--ptxas-options=-v` to check)
- automatic array variables (in the absence of qualifiers) reside in thread-local memory
- the type of memory used will be crucial for the performance of the application

Elementary data transfers with CUDA

Memory allocation

Arrays in device global memory are typically allocated from CPU code. Functions:

- `cudaMalloc(void ** pointer, size_t nbytes)`
- `cudaMemset(void * pointer, int value, size_t count)`
- `cudaFree(void* pointer)`

```c
int n = 1024;
int nbytes = 1024*sizeof(int);
int *a_d = 0;
cudaMalloc( ( void**) &a_d , nbytes );
cudaMemset( a_d , 0 , nbytes);
cudaFree(a_d);
```

Data transfers

The elementary function for data transfers is

```c
void cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);
```
- direction is one of cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost or cudaMemcpyDeviceToDevice and specifies location of src and dst
- blocks CPU thread (asynchronous transfers possible in streams)
CUDA Programming

Kernel execution

Function qualifiers
- __global__ void f()
  - function called from host, executed on device
  - must return void
- __device__ int f()
  - function called from device, executed on device
- __host__ int f()
  - function called from host, executed on device

All __global__ and __device__ functions have the following automatic variables:
- dim3 gridDim; — dimension of the grid in blocks
- dim3 blockDim; — dimension of the block in threads
- dim3 blockIdx; — block index within grid
- dim3 threadIdx; — thread index within block

The indices can be used to construct a global thread index, for instance for a block size of 5 threads,

\[
\text{thread\_index} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x};
\]

CUDA Programming

Kernel execution

Execution configuration

Modified C function call syntax:

\[
\text{kernel}<<<\text{dim3} \, \text{dG}, \, \text{dim3} \, \text{dB}>>>(...);
\]

Execution configuration:
- dG: dimension and size of grid in blocks
  - two-dimensional, \(dG.x\) and \(dG.y\)
  - total number of blocks launched is \(dG.x \times dG.y\)
- dB: dimension and size of each block
  - two- or three-dimensional, \(dB.x, dB.y, \text{and} \, dB.z\)
  - total number of threads per block is \(dB.x \times dB.y \times dB.z\)
  - if not specified, \(dB.z = 1\) is assumed

CUDA Programming

Memory hierarchy

Per thread
- Registers (extra fast, no copy for ops)
- Local memory

Thread blocks: shared memory
- allocated by thread block, same lifetime as block
- allocate as __shared__ int shared_array[DIM];
- low latency (of the order of 10 cycles), bandwidth up to 1 TB/s
- use for data sharing and user-managed cache

Per device: global memory
- accessible to all threads on device
- lifetime is user-defined

\[
\text{cuda\_malloc}(\text{void **} \, \text{pointer}, \text{size_t} \, \text{nbytes});
\]

\[
\text{cuda\_free}(\text{void*} \, \text{pointer});
\]
- latency several hundred clock cycles
- bandwidth \(\approx 160\) GB/s on Fermi
- access pattern needs to conform to coalescence rules for good performance

Per host: device memory
- no direct access from CUDA threads
- copy data to/from device with

\[
\text{cuda\_Memcpy}(\text{void*} \, \text{dest}, \text{void*} \, \text{src}, \text{size_t} \, \text{nbytes}, \text{cuda\_MemcpyHostToDevice});
\]
More generally, the different types of memory have the following characteristics:

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Cached</th>
<th>Access</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>On-chip</td>
<td>N/A</td>
<td>R/W</td>
<td>One thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Local</td>
<td>Off-chip</td>
<td>No</td>
<td>R/W</td>
<td>One thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>N/A</td>
<td>R/W</td>
<td>All threads in a block</td>
<td>Block</td>
</tr>
<tr>
<td>Global</td>
<td>Off-chip</td>
<td>(Yes)</td>
<td>R/W</td>
<td>All threads + host</td>
<td>Application</td>
</tr>
<tr>
<td>Constant</td>
<td>Off-chip</td>
<td>Yes</td>
<td>R</td>
<td>All threads + host</td>
<td>Application</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>Yes</td>
<td>R</td>
<td>All threads + host</td>
<td>Application</td>
</tr>
</tbody>
</table>

Unified virtual addressing

- **System Memory**
- **GPU0 Memory**
- **GPU1 Memory**

Direct GPU data transfers

- GPUDirect 1.0: direct GPU memory access by devices such as network adaptors
- GPUDirect 2.0: direct copies from GPU to GPU inside a node
- GPUDirect/CUDA 5.0: direct communication between GPUs in different nodes
- MPI integration under development
CUDA distribution

CUDA is available together with documentation and libraries on CUDA zone, https://developer.nvidia.com/cuda-zone

Current version is CUDA 10.

Important tools:
- CUDA compiler, nvcc
- CUDA visual profiler
- NVIDIA Nsight
- CUDA gdb
- OpenACC

Additional libraries:
- cuRAND
- cuFFT
- nvGRAPH, ...

CUDA compiler invocation:

Simplest case
nvcc -o test test.cu

Some options
nvcc -arch=sm_35 -rdc=true -I./Random123/include --ptxas-options=-v ising2D.cu -o ising2D

Summary and outlook

This lecture
This lecture has given a basic introduction into GPGPU and, in particular, the CUDA framework of GPU programming. Some basic examples hopefully provided a feel for how to go about in using these devices for scientific computing.

Next lecture
In lecture 2, we will start using GPUs for simulating spin models with local algorithms. In terms of GPU programming, a number of additional concepts such as thread synchronization, memory coalescence, and atomic operations will be introduced.

Reading