VT Summer 2013

Andrey Asadchev

VT

July 24, 2013

Outline

- Accelerators overview
- OpenMP, OpenAcc, OpenCL, CUDA
- NVidia CUDA overview
- Setting up shell
- Discovering CUDA devices
- Device resources and occupancy
- CUBLAS DGEMM vs MKL
- CUDA hash function
- Debugging
- CUDA memory
- CUDA reduction kernel
- Profiling
- CUDA/CUBLAS streams
- Simple tensor contraction

Accelerators overview

Intel MIC/Xeon Phi:

- X86 compatible
- 512-bit SIMD
- Around 1TFlops DP
- Can be programmed using OpenMP
- Linux OS - a headless "node"

NVidia CUDA (Tesla, *Fermi*, Kepler)

- PTX - specialized ISA
- 16 32-core processors
- Over 1TFlops DP for Kepler
- No *viable* OpenMP support

OpenMP, OpenAcc, OpenCL

- OpenMP
  - Widely used for parallelizing programs
  - Parallel code annotated using `#pragma omp v`
  - Virtually supported by every compiler
  - Supported on MIC but not on GPUs
  - Doesn't require code rewrite but may need code restructuring
- OpenAcc
  - OpenMP-like set of `#pragma openacc`
  - Limited support (Cray and PGI)
  - In principle doesn't require code rewrite but ...
- OpenCL
  - C-like language for writing and executing kernels in heterogenous environments
  - Kernel compilation is handled by runtime library
  - JIT-like - can generate kernels on the fly
  - Requires extensive code rewrite

CUDA

- Parallel programming and runtime platform from NVidia
- Most often associated with CUDA C/C++ (C++ like language)
- Support for templates and objects
- Needs NVCC compiler
- Requires extensive code rewrite
- Also CUDA Fortran from PGI

Majority of time will be spent in CUDA C++

CUDA Terminology

- Thread - the smallest grain of work, e.g. a single loop iteration. NOTHING like CPU thread.
- Warp - a set of (32) consecutive threads, think 32-wide SIMD vector. The actual hardware is 2x16-wide SIMD but from programmer viewpoint it looks like warp-wide SIMD.
- Thread Block - group of threads executed together, a larger grain of work.
  Threads in block can be syncronized and communicate via shared memory
- Grid - a group of blocks, constitutes the entire work. Limited syncronization among blocks.

Thread and block have unique x,y,z index: `threadIdx, blockIdx`
Blocks and grids have x,y,z dimensions: `blockDim, gridDim`

Setting up shell

First, set up head node shell

```
hpc05@hs162:~# tar -xzf /home/hpc05/gpu.tar.gz
hpc05@hs162:~# source gpu/source.me
hpc05@hs162:~# cd gpu
hpc05@hs162:~/gpu#
hpc05@hslogin1:~/gpu# make devices
g++ -g -std=c++0x ... devices.cpp -o devices ... -lcudart
```

Next, set up compute node shell in another terminal

```
hpc05@hslogin1:~/gpu# qsub -I
qsub: waiting for job 13368.master.cluster to start
qsub: job 13368.master.cluster ready
hpc05@hs162:~# source gpu/source.me
hpc05@hs162:~# cd gpu
hpc05@hs162:~/gpu# ./devices
```

Discovering CUDA devices

- CUDA API is available online
  `http://docs.nvidia.com/cuda/cuda-runtime-api/`
- Navigate to Device Management
- On head node, open `./devices.cpp` with your favorite Emacs editor
- On compute node, see the output of `./devices` command

Some important properties are:

- totalGlobablMem - slow device/global memory size
- sharedMemPerBlock - fast shared memory per SM
  Limits how many blocks can be active at a time. Current max is 8
- regsPerBlock - fast private 32-bit registers per SM
  Limits how many threads can be active at a time.

Device resources and occupancy

- Access to global memory is very slow - high latency
- While warp is waiting for memory transfer to complete, another warp can execute
- Provided enough warps can execute (so called active warps), latency can be hidden
- If maxThreadsPerMultiProcessor is 1536 and warpSize is 32:
  the maximum number of active warps is 48
  the maximum number of active blocks is 8
- If each thread needs 32 registers, then max occupancy is $(32768/32)/1536 = 0.666$
- If each block needs 8KB of shared memory, then max occupancy is $(49152/8192)/8 = 0.75$

DGEMM example

Goals:

- Mix C++ and calls to CUDA *runtime*.
- CUDA error checking
- Performance API (PAPI)
- CUDA profiler

```
make dgemm
export CUDA_PROFILE=1
export COMPUTE_PROFILE_CONFIG=cuda-profile.config
./dgemm
less cuda_profile_0.log
unset CUDA_PROFILE
nvprof ./dgemm
```

Hash function example

Goals:

- Mix C++ and CUDA.
- Simple CUDA coding
- Atomic ops
- CUDA debugging
  - cuda-gdb
  - cuda-memcheck
  - printf
  - assert

Thrust

http://docs.nvidia.com/cuda/thrust/

```
#include <thrust/device_ptr.h>
#include <thrust/device_malloc.h>
#include <thrust/device_free.h>
#include <thrust/copy.h>
...
thrust::device_ptr<int> dev_ptr = thrust::device_malloc<int
thrust::copy(host.data(), host.data()+N, dev_ptr);
thrust::device_free(dev_ptr);
...
```

Tensor Contraction

Assume i, j, k are occupied orbitals and a, b, e are virtual
Compute $t(i, j, a, b) - = u(j, k, e, a) * v(i, k, e, b)$
with constraint that memory scales as O^2V

```
// Pseudocode
for b in V {
    v(i,ke) = v(i,k,e,b); // load v tile
    // compute t(i,j,a)
    for a in V {
        u(j,ke) = u(j,k,e,a); // load u tile
        t(i,j,a) = dgemm(v, u');
    }
    store t(i,j,a);
}
```

Try to implement in `tensor.cpp`

Resources

- http://docs.nvidia.com/cuda/cuda-c-programming-guide/
- http://stackoverflow.com/questions
- http://www.drdobbs.com/parallel
- http://google.com