GPU Acceleration

Julien Demouth
Outline

- CUDA Refresher
- Kepler Architecture
  - SMX
- CUDA 5 Features
  - CUDA Dynamic Parallelism
  - Hyper-Q
  - Device-Code Linker
3 Ways to Accelerate Applications

Applications

- Libraries
  - "Drop-in" Acceleration
- OpenACC Directives
  - Easily Accelerate Applications
- Programming Languages
  - Maximum Flexibility
GPU Accelerated Libraries

- NVIDIA cuBLAS
- NVIDIA cuRAND
- NVIDIA cuSPARSE
- NVIDIA NPP
- GPU VSIPL
- CULA tools
- MAGMA
- NVIDIA cuFFT
- Rogue Wave Software
- CUSP
- libjacket
- Thrust

Vector Signal Image Processing
GPU Accelerated Linear Algebra
Matrix Algebra on GPU and Multicore
Sparse Linear Algebra
Building-block Algorithms for CUDA
C++ STL Features for CUDA
OpenACC Directives

Program myscience
... serial code ...
$acc kernels
  do k = 1,n1
  do i = 1,n2
  ... parallel code ...
  enddo
endo
d$acc end kernels
... 
End Program myscience

CPU

GPU

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

Your original Fortran or C code
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);

__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);
Opening the CUDA Platform with LLVM

CUDA compiler source now available with Open source LLVM Compiler

SDK includes specification documentation, examples, and verifier

Provides ability for anyone to add CUDA to new languages and processors

Learn more at http://developer.nvidia.com/cuda-source
GPUs are Mainstream

- Oil & Gas
- Edu/Research
- Government
- Life Sciences
- Finance
- Manufacturing

- Schlumberger
- BR
- PETROBRAS
- Chevron
- Total
- Paradigm
- Chinese Academy of Sciences
- Georgia Tech
- Harvard School of Engineering and Applied Sciences
- BAE Systems
- Air Force Research Laboratory
- NASA
- Naval Research Laboratory
- Boston Scientific
- MGH 1811
- Max Planck Institute
- Bloomberg
- BNP PARIBAS
- STANDARD LIFE
- J.P. Morgan
- Agilent
- ANSYS
- Autodesk
- NumeriX
- ACUSIM SOFTWARE
KEPLER ARCHITECTURE
The Kepler GPU

Performance

Efficiency

Programmability
**Tesla K10**

- 3x Single Precision
- 1.8x Memory Bandwidth
- Image, Signal, Seismic

**Available Now**

**Tesla K20**

- 3x Double Precision
- Hyper-Q, Dynamic Parallelism
- CFD, FEA, Finance, Physics

**Available Q4 2012**
Kepler GK110 Block Diagram

Architecture
- 7.1B Transistors
- 15 SMX units
- > 1 TFLOP FP64
- 1.5 MB L2 Cache
- 384-bit GDDR5
## SMX Balance of Resources

<table>
<thead>
<tr>
<th>Resource</th>
<th>Kepler GK110 vs Fermi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating point throughput</td>
<td>2-3x</td>
</tr>
<tr>
<td>Max Blocks per SMX</td>
<td>2x</td>
</tr>
<tr>
<td>Max Threads per SMX</td>
<td>1.3x</td>
</tr>
<tr>
<td>Register File Bandwidth</td>
<td>2x</td>
</tr>
<tr>
<td>Register File Capacity</td>
<td>2x</td>
</tr>
<tr>
<td>Shared Memory Bandwidth</td>
<td>2x</td>
</tr>
<tr>
<td>Shared Memory Capacity</td>
<td>1x</td>
</tr>
</tbody>
</table>
New ISA Encoding: 255 Registers per Thread

- Fermi limit: 63 registers per thread
  - A common Fermi performance limiter
  - Leads to excessive spilling

- Kepler: Up to 255 registers per thread
  - Especially helpful for FP64 apps

- Ex. Quda QCD fp64 sample runs 5.3x faster
  - Spills are eliminated with extra registers
New High-Performance SMX Instructions

SHFL (shuffle) -- Intra-warp data exchange

ATOM -- Broader functionality, Faster

Compiler-generated, high performance instructions:
- bit shift
- bit rotate
- fp32 division
- read-only cache
**Texture Cache Unlocked**

- Added a new path for compute
  - Avoids the texture unit
  - Allows a global address to be fetched and cached
  - Eliminates texture setup
- Why use it?
  - Separate pipeline from shared/L1
  - Highest miss bandwidth
  - Flexible, e.g. unaligned accesses
- Managed automatically by compiler
  - “const __restrict” indicates eligibility
CUDA DYNAMIC PARALLELISM
Improving Programmability

- Library Calls from Kernels
- Simplify CPU/GPU Divide
- Batching to Help Fill GPU
- Dynamic Load Balancing
- Data-Dependent Execution
- Recursive Parallel Algorithms

Dynamic Parallelism

Programmability

Occupancy

Execution
What Does It Mean?

**GPU as Co-Processor**

**Autonomous, Dynamic Parallelism**
CDP Nested Launch

- Can create new grids at run-time
- Composable: can only launch into own position in stream
- Can yield until new work completes, then resume using result (i.e. join)
- Launch by and thread is visible to thread’s whole block

```c
int main() {
    float *data;
    setup(data);
    A <<< ..., stream >>> (data);
    B <<< ..., stream >>> (data);
    C <<< ..., stream >>> (data);
    cudaThreadSynchronize();
    return 0;
}
```

```c
__global__ void B(float *data) {
    do_stuff(data);
    X <<< ..., stream >>> (data);
    Y <<< ..., stream >>> (data);
    Z <<< ..., stream >>> (data);
    cudaStreamSynchronize(stream);
    do_more_stuff(data);
}
```
Fermi allows 16-way concurrency

- Up to 16 grids can run at once
- But CUDA streams multiplex into a single queue
- Overlap only at stream edges
Kepler Improved Concurrency

Kepler allows 32-way concurrency

- One work queue per stream
- Concurrency at full-stream level
- No inter-stream dependencies
HYPER-Q + MPI PROXY
Hyper-Q + Proxy Enable Better Utilization

- Multiple MPI ranks can execute concurrently
- Use as many MPI ranks as in the CPU-only case
- Particularly interesting for strong scaling
Hyper-Q + Proxy from the User’s Perspective

- No code modifications needed
- Proxy server on host
  - All processes communicate with the GPU via proxy
    - `nvidia-cuda-proxy-control -d`
- Ideal for applications with:
  - MPI-everywhere
  - Non-negligible CPU work
  - Partially migrated to GPU
GPU DIRECT
Kepler Enables Full NVIDIA GPUDirect™
CUDA Compilation

- Separation of host and device code
- Device code translates into:
  - Device-specific binary (.cubin)
  - Device-independent assembly (.ptx)
- Device code embedded in host object data
CUDA 5 Introduces Device Code Linker

- foo.cu
  - foo.ptx
  - foo.c
    - foo.cubin
  - foo.o
    - foo.cubin (JIT)
  - a.out
    - foo.cubin (JIT)
    - bar.cubin (JIT)

- bar.cu
  - bar.ptx
  - bar.c
    - bar.cubin

Device Linker
Host Linker
Device Linker Invocation

- Introduction of an optional link step for device code
  nvcc -arch=sm_20 -dc a.cu b.cu
  nvcc -arch=sm_20 -dlink a.o b.o -o link.o
  g++ a.o b.o link.o -L<path> -lcudart

- Link device-runtime library for dynamic parallelism
  nvcc -arch=sm_35 -dc a.cu b.cu
  nvcc -arch=sm_35 -dlink a.o b.o -lcudadevrt -o link.o
  g++ a.o b.o link.o -L<path> -lcudadevrt -lcudart

- Currently, link occurs at cubin level (PTX not supported)
EXASCALE PATH
Echelon Team
Power is the Problem

1. Data Movement Dominates Power
2. Optimize the Storage Hierarchy
3. Tailor Memory to the Application
The High Cost of Data Movement

Fetching operands costs more than computing on them.

- 64-bit DP: 20 pJ
- 256-bit buses: 26 pJ
- 256-bit access: 50 pJ
- 8 kB SRAM: 50 pJ
- DRAM Read/Write: 256 pJ
- Efficient off-chip link: 1 nJ
- 28 nm Silicon Technology: 16 nJ
- Total: 500 pJ

20 mm
Echelon Architecture (1/2)

Lane — DFMAs, 20 GFLOPS

SM — 8 lanes, 160 GFLOPS

Chip — 128 SMs, 20.48 TFLOPS + 8 Latency Processors

Node MCM — 20 TFLOPS + 256 GB
Cabinet – 128 Nodes – 2.56 PF – 50 KW
Central Router Module(s), Dragonfly Interconnect

System – 400 Cabinets – 1 EF – 20 MW
Dragonfly Interconnect