Basics of CADA Programming
- CUDA 4.0 and newer

Feb 19, 2013
Outline

• CUDA basics
  • Extension of C
  • Single GPU programming
  • Single node multi-GPUs programming
• A brief introduction on the tools
  • Jacket
  • CUDA FORTRAN – PGI compiler
• Hands-on exercises
CUDA - Compute Unified Device Architecture

• General-Purpose Programming Model
• Standalone driver to load computation programs into GPU Graphics-free API
• Data sharing with OpenGL buffer objects
• Easy to use and low-learning curve

• CUDA allows developers to use C
  • Also supports other languages, such as FORTRAN, DirectCompute, OpenCL, OpenACC.
Survey Questionnaires:

Why are you interested in GPU computing?

What kind of applications do you need to accelerate on GPU hardware?

Do you have the computing code(s) already on CPU? If yes, in what language is it written (C, FORTRAN or Matlab)?

Do you have a deadline or milestone to get your computing on GPU hardware? When?

Specific need about the hardware (memory, multi-GPU and interconnect need)?

Will you learn CUDA or use tools to accelerate your calculations on GPU hardware?

How can we do better for the future GPU workshop: Specific topics are you interested? Specific acceleration tools?
GPUs can be controlled by:  
– A single CPU thread  
– Multiple CPU threads belonging to the same process  
– Multiple CPU processes

Definitions used:  
– CPU process has its own address space  
– A process may spawn several threads, which can share address space
Figure 1.2: This figure is inspired by the figure found in *Understanding the CUDA Data Parallel Threading Model: A Primer* written by Michael Wolfe from The Portland Group.

Deeply cached memory hierarchy. GPUs from NVIDIA have many processors, what they call streaming processors (SP). Each streaming processor is capable of executing a sequential thread. For a GPU with Fermi architecture, like the one we are using, every 32 streaming processors is organized in a Streaming Multiprocessor (SM). A GPU can have one or more multiprocessors on board. For example, the Tesla C2075 GPU card we are using, has 14 multiprocessors built in. Except for 32 streaming processors, each multiprocessor also equipped with 2 warp scheduler, 2 special function units (4 in some GPUs), a set of 32-bit registers and 64KB of configurable shared memory. Warp scheduler is responsible for threads control; SFU handles transcendentals and double-precision operations. For a GPU with Kepler architecture, every 192 streaming processors is organized in a multiprocessor. There are also more warp schedulers and SFUs built in. Shared memory, or L1 cache, is a small data cache that can be configured through software. Shared memory is also shared among all the streaming processors within one multiprocessor. Compared with on-board memory, shared memory is low-latency (usually register speed) and has high bandwidth. Each multiprocessor has 64KB of shared memory that can be configured by user using special commands in host code. Share memory is distributed to software-managed data cache and hardware data cache. User can choose to assign either 48KB to software-managed data cache (SW) and 16KB to hardware data cache (HW) or the other way around.

1.5 How does CUDA connect with hardware? When the host code invokes a kernel grid through CUDA program, blocks in the grid are distributed to different multiprocessors based on available execution capacity of each multiprocessor. Each multiprocessor is capable of process one or more blocks throughout the kernel execution. However, each block can only be processed by one multiprocessor. Fermi architecture supports up to 48 active warps on each multiprocessor. The advantage of having many active warps in process at the same time is the significantly reduction of memory latency. Traditionally, memory latency is reduced by adding more cache memory hierarchy into the system. However, by using high degree of multithreading, 1.5. How does CUDA connect with hardware? 3
GPU Device Computing Components

- Streaming Processors (SP). Each streaming processor is capable of executing a sequential thread, also called a GPU core.
- A number of streaming processors is organized in a Streaming Multiprocessor (SM).
- A warp in CUDA, then, is a group of 32 threads, which is the minimum size of the data processed in **SIMD** fashion by a CUDA multiprocessor.
GPU Device Computing Components (continued)

• Each multiprocessor also equipped with
  • warp scheduler - responsible for threads control
  • special function units (SFU) - transcendental operations and double-precision
    operations
  • a set of 32-bit registers
  • 64KB of configurable shared memory.

• A GPU device has one or more multiprocessors on board.
  e.g, Tesla C2075 GPU card, has 14 multiprocessors
• Newer device has more SMs, SFUs, or more complicated
  architecture.
## GPU Device Memory Hierarchy

<table>
<thead>
<tr>
<th>Memory</th>
<th>Scope of Access</th>
<th>Lifetime</th>
<th>R/W ability</th>
<th>Speed</th>
<th>Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Thread</td>
<td>Kernel</td>
<td>R/W</td>
<td>Fast</td>
<td>Automatic Variables</td>
</tr>
<tr>
<td>Local</td>
<td>Thread</td>
<td>Kernel</td>
<td>R/W</td>
<td>Fast</td>
<td>Automatic Arrays</td>
</tr>
<tr>
<td>Shared</td>
<td>Block</td>
<td>Kernel</td>
<td>R/W</td>
<td>Fast</td>
<td><strong>shared</strong></td>
</tr>
<tr>
<td>Global</td>
<td>Grid</td>
<td>Host</td>
<td>R/W</td>
<td>Slow</td>
<td><strong>device</strong></td>
</tr>
<tr>
<td>Constant</td>
<td>Grid</td>
<td>Host</td>
<td>Read only</td>
<td>Fast</td>
<td><strong>constant</strong></td>
</tr>
</tbody>
</table>
CUDA Software Environment

New Syntax:

```c
<<< ... >>> /* kernel or executable, will run on GPU device
__host__, __global__, __device__, __constant__, __shared__,
__device__ __syncthreads()
```

Built-in Variables:

- **dim3** `gridDim`;
  - Dimensions of the grid in blocks
- **dim3** `blockDim`;
  - Dimensions of the block in threads
- **dim3** `blockIdx`;
  - Block index within the grid
- **dim3** `threadIdx`;
  - Thread index within the block
CUDA Software Environment

Restriction Relax:
Device with compute capability 2.0 or higher supports:
- recursion in device code
- branching
- function pointers
but efficiency may not be great.

CUDA API/Libraries
CUDA Runtime (Host and Device)
Device Memory Handling (cudaMalloc,...)
Built-in Math Functions (sin, sqrt, mod, ...)
Atomic operations (for concurrency)
Data-types (dim2, dim3, ...)
CUDA Software Environment

Function Type Qualifiers

Specify whether a function executes on the host or on the device and whether it is callable from the host or from the device.

__global__ a kernel is executed on the device, callable from the host only. __global__ functions must have void return type.

Any call to a __global__ function must specify its execution configuration, i.e., <<< ... >>>

A call to a __global__ function is asynchronous; returns before the device has completed its execution.

__device__ a function is executed on and callable from the device only.

__host__ a function that is executed on the host and callable from the host only.
CUDA Software Environment

`__global__` and `__host__` qualifiers cannot be used together for one function.

`__device__` and `__host__` qualifiers can be used together with `__CUDA_ARCH__` macro to differentiate code paths:

```c
__host__ __device__ func()
{
    #if __CUDA_ARCH__ == 100
        // Device code path for compute capability 1.0
    #elif __CUDA_ARCH__ == 200
        // Device code path for compute capability 2.0
    #elif __CUDA_ARCH__ == 300
        // Device code path for compute capability 3.0
    #elif !defined(__CUDA_ARCH__)
        // Host code path
    #endif
}
```

CUDA 4.0 – software
Compute capability – hardware
Steps of GPU computing under CUDA

1. Copy data from main mem to GPU’s
2. CPU instructs the process to GPU
3. **GPU executes parallel in each core**
4. Copy the result back to CPU mem

int main() {
    int N = 10000;
    size_t size = N * sizeof(float);
    // Allocate input vectors h_A, h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    // and initialize them.
    ........

    // Allocate vectors in device memory
    float* d_A; cudaMalloc(&d_A, size);
    float* d_B; cudaMalloc(&d_B, size);
    float* d_C; cudaMalloc(&d_C, size);
    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    VecAdd<<<1,N>>>(d_A, d_B, d_C, N);
    // Copy result from device memory to host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    // Free host memory ...
    Free(h-A); }

    // Device code
    __global__ void VecAdd(float* A, float* B, float* C, int N) {
        int i = blockDim.x * blockIdx.x + threadIdx.x;
        if (i < N)
            C[i] = A[i] + B[i];
    }

Key difference from CPU computing

// on CPU computing
// VecAdd(h_A, h_B, h_C, N);
// OMP_NUM_THREADS

// GPU computing – thread hierarchy

int numBlocks = 1;
dim3 threadsPerBlock(N);

VecAdd<<< NumBlocks, ThreadPerBlock >>>(d_A, d_B, d_C, N);
VecAdd<<< 1, N >>>(d_A, d_B, d_C, N);

// <<< NumBlocks, ThreadPerBlock>>>
// how to map the available cores to number of thread

// Device code
__global__ void VecAdd(float* A, float* B, float* C, in
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
Device Thread Hierarchy

Dim3 threadIdx;
    // Built-in 3-D variable for the efficiency of accessing memory
    // threadIdx.x, threadIdx.y, threadIdx.z

For a 1-D block, a linear mapping of cores to threads
For a 2D block of size (Dx, Dy),
    the thread ID of a thread of index (x, y) is
        (x + y Dx);
For a 3D block of size (Dx, Dy, Dz),
    the thread ID of a thread of index (x, y, z) is
        (x + y Dx + z Dx Dy).
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main() {
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd <<< numBlocks, threadsPerBlock >>> (A, B, C); ... }
CUDA Features Useful for MultiGPU

• Control multiple GPUs with a single GPU thread
  – Simpler coding: no need for CPU multi-threading
• Peer-to-Peer (P2P) GPU memory copies
  – Transfer data between GPUs using PCIe P2P support
  – Done by GPU DMA hardware – host CPU is not involved
• Data traverses PCIe links, without touching CPU memory
  – Disjoint GPU---pairs can communicate simultaneously
• Streams:
  – executing kernels and memcopies concurrently
  – Up to 2 concurrent memcopies: to/from GPU
• P2P exception: 2GPUs connected to different IOH chips
  – IOH (Intel I/O Hub chip on motherboard) connected via QPI
• QPI and PCIe don’t agree on P2P protocol
  – CUDA API will stage the data via host memory
This is the HW configuration of the PLX cluster.
CUDA and multi-GPU programming

• Single-process / multiple GPUs:
  - Unified virtual address space
  - Ability to directly access peer GPU's data
  - Ability to issue P2P mem copies
    # No staging via CPU memory
    # High aggregate throughput for many-GPU nodes

• Multiple-processes:
  - Direct to maximize performance when both PCIe and IB transfers are needed

• Streams and asynchronous kernel/copies
  - Allow overlapping of communication and execution
  - Applies whether using single- or multiple processes to control GPUs
Use of multiple GPUs on the same node

```c
int devs = 4

for (int d=0; d < devs; d++)
{
    cudaSetDevice(d);
    cudaMalloc((void**)&d_A, size); cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
}
```
GPU acceleration tools

Jacket - Wraps some of Matlab codes for enhancing their performance by running on GPU

module load jacket matlab
matlab
>> gactivate
>> ghelp % list all functions supported by Jacket
>> ghelp try %

All Jacket functions may be found at:
How can Jacket help?

Partial support - Not every Matlab calculation can benefit

Hot spot – part of the code consumes most of the CPU time

Special functions and toolbox – are they being used? Are they supported by Jacket?

If yes, modify the code according to Jacket's syntax.
Use of Jacket

- replacement of low-level MATLAB data structures
- GPU computation and acceleration

<table>
<thead>
<tr>
<th>Jacket Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSINGLE</td>
<td>Casts a CPU matrix to a single precision floating point GPU matrix.</td>
<td>A = gsingle(B);</td>
</tr>
<tr>
<td>GDOUBLE</td>
<td>Casts a CPU matrix to a double precision floating point GPU matrix.</td>
<td>A = gdouble(B);</td>
</tr>
<tr>
<td>GLOGICAL</td>
<td>Casts a CPU matrix to a binary GPU matrix. All non zero values are set to '1'. The input matrix can be a GPU or CPU datatype.</td>
<td>A = glogical(B); A = glogical(0:4);</td>
</tr>
<tr>
<td>GINT8, GIINT8, GINT32, GIINT32</td>
<td>Cast a CPU matrix to a signed and unsigned 8-bit or 32-bit integer GPU matrix respectively.</td>
<td>A = gint8(B); A = guint8(B); A = gint32(B); A = guint32(B);</td>
</tr>
<tr>
<td>GZEROS, ZEROS</td>
<td>Create a matrix of zeros on the GPU.</td>
<td>A = gzeros(5,'double'); A = zeros(2,6,gdouble);</td>
</tr>
<tr>
<td>GONES, ONES</td>
<td>Create a matrix of ones on the GPU.</td>
<td>A = gones(5,'double'); A = ones([3 9], gdouble);</td>
</tr>
<tr>
<td>GEYE</td>
<td>Creates an identity matrix on the GPU.</td>
<td>A = geye(5);</td>
</tr>
<tr>
<td>GRAND or RAND</td>
<td>Creates a random matrix on the GPU, with uniformly distributed pseudorandom numbers.</td>
<td>A = grand(5,'double'); A = rand(5,gdouble);</td>
</tr>
<tr>
<td>GRANDN</td>
<td>Creates a random matrix on the GPU, with normally distributed pseudorandom numbers.</td>
<td>A = grandn(5,'double'); A = randn(5,gdouble);</td>
</tr>
</tbody>
</table>
## Basic functions

<table>
<thead>
<tr>
<th>Jacket Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>GHELP</td>
<td>Retrieve information on the Jacket</td>
<td><code>ghelp sum;</code></td>
</tr>
<tr>
<td>GACTIVATE</td>
<td>Used for manual activation of a Jacket license.</td>
<td><code>gactivate;</code></td>
</tr>
<tr>
<td>GSELECT</td>
<td>Select or query which GPU is in use.</td>
<td><code>gselect(0);</code></td>
</tr>
<tr>
<td>GFOR</td>
<td>Executes FOR loop in parallel on GPU.</td>
<td><code>% loop body</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GCOMPILE</td>
<td>Compile M-code directly into a single CUDA kernel.</td>
<td><code>gprofile on; foo; gprofile off;</code></td>
</tr>
<tr>
<td>GPROFILE</td>
<td>Profile code to compare CPU versus GPU runtimes.</td>
<td><code>gprofview;</code></td>
</tr>
<tr>
<td>GPROFVIEW</td>
<td>Visual representation of profiling data.</td>
<td><code>geval;</code></td>
</tr>
<tr>
<td>GEVAL</td>
<td>Evaluate computation and leave results on GPU.</td>
<td><code>gsync(A);</code></td>
</tr>
<tr>
<td>GSYNC</td>
<td>Block until all queued GPU computation is complete.</td>
<td><code>gcachefilename.m');</code></td>
</tr>
<tr>
<td>GCACHE</td>
<td>Save GPU compiled code for given script.</td>
<td><code>gsave('filename', A);</code></td>
</tr>
<tr>
<td>GLOAD</td>
<td>Load from disk directly into the GPU. Requires the Jacket SDK.</td>
<td><code>gread('filename', OFFSET, DATA);</code></td>
</tr>
<tr>
<td>GSAVE</td>
<td>Save data to disk as text file directly from the GPU.</td>
<td><code>gplot(A);</code></td>
</tr>
<tr>
<td>GREAD</td>
<td>Load from disk directly into the GPU, with option to specify the byte range. Requires the Jacket SDK.</td>
<td><code>gwrite('filename', OFFSET, DATA);</code></td>
</tr>
</tbody>
</table>

**Graphics**

Library Functions contained in the Graphics Library.
CUDA FORTRAN – PGI compiler

1. A small set of extensions to Fortran
2. Supports and is built up on the CUDA
3. A lower-level explicit programming model
4. Substantial run-time library components
5. An analog to NVIDIA's CUDA C
6. compiler Portland License!

module load pgi
pgfortran -Mcuda Sgemm.F90 -lcublas
/usr/bin/time ./a.out < input
Hands-on exercises

To login to the gpu nodes, type one of the following

    vglconnect gput02
    vglconnect gput03
    vglconnect gput04

To get the hands-on exercises:

    cp /scratch/.
    tar
Hands-on exercises

To login to the gpu nodes, type one of the following

    vglconnect gput02
    vglconnect gput03
    vglconnect gput04

To get the hands-on exercises:
    cp /scratch/.
    tar
Hands-on exercises

Write a cuda program to implement a vector addition using one GPU and compare its results with CPU implementation.

Write a cuda program to implement a vector addition using 4 GPUs and compare its results with four CPU implementation

https://www.msi.umn.edu/tutorials/gpu

To login to the gpu nodes, type one of the following

  vglconnect -s gput02
or  vglconnect –s gput03
or  vglconnect –s gput04

To get the hands-on exercises:

  cp /scratch/wrkshp_Feb19_2013.tar .
  tar –xvf wrkshp_Feb19_2013.tar
Hands-on exercises

To compile:
  cd gpuwksp
  module load cuda
  make clean
  make

To run a job in the included cases
  C/bin/linux/release/deviceQuery

To run graphic simulation
  vglrun C/bin/linux/release/nbody -numdevices=N
  where N not exceeds 4