Basics of CADA Programming
- CUDA 4.0 and newer

Feb 19, 2013

•Supercomputing Institute
•for Advanced Computational Research

University of Minnesota
Driven to Discover℠

Outline

- CUDA basics
    - Extension of C
    - Single GPU programming
    - Single node multi-GPUs programing
- 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?

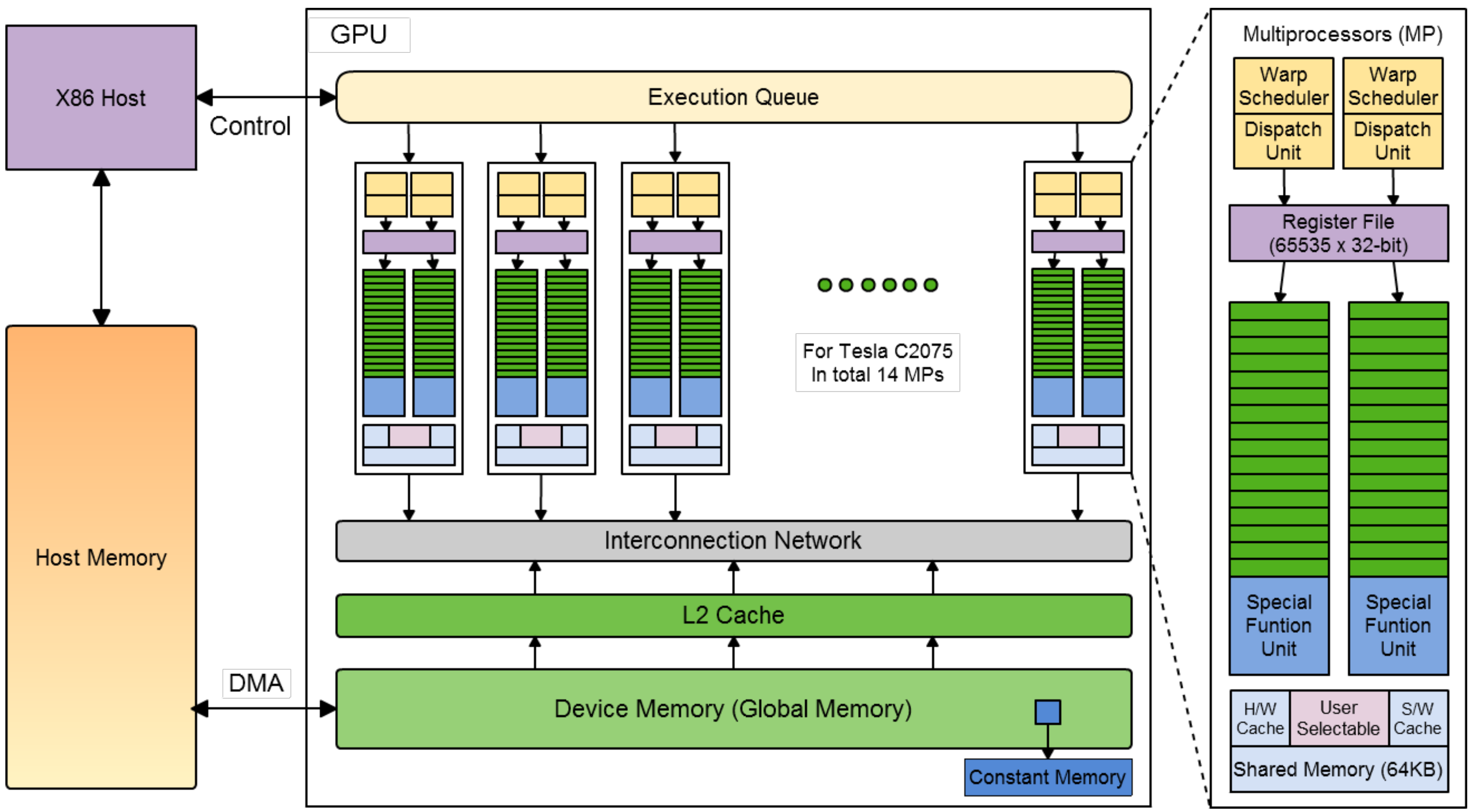
Specific need about the hardware (memory, mutli-GPU and interconnect need)?

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


X86 Host
Control
Host Memory
DMA
GPU
Execution Queue
For Tesla C2075
In total 14 MPs
Interconnection Network
L2 Cache
Device Memory (Global Memory)
Constant Memory
Multiprocessors (MP)
Warp Scheduler
Dispatch Unit
Warp Scheduler
Dispatch Unit
Register File (65535 x 32-bit)
Special Funtion Unit
Special Funtion Unit
H/W Cache
User Selectable
S/W Cache
Shared Memory (64KB)

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) - transcendentals 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

Memory	Scope of Access	Lifetime	R/W ability	Speed	Declaration
Register	Thread	Kernel	R/W	Fast	Automatic Variables
Local	Thread	Kernel	R/W	Fast	Automatic Arrays
Shared	Block	Kernel	R/W	Fast	**__shared__**
Global	Grid	Host	R/W	Slow	**__device__**
Constant	Grid	Host	Read only	Fast	**__constant__**

CUDA Software Environment

New Syntax:

<<< ... >>> /* 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.

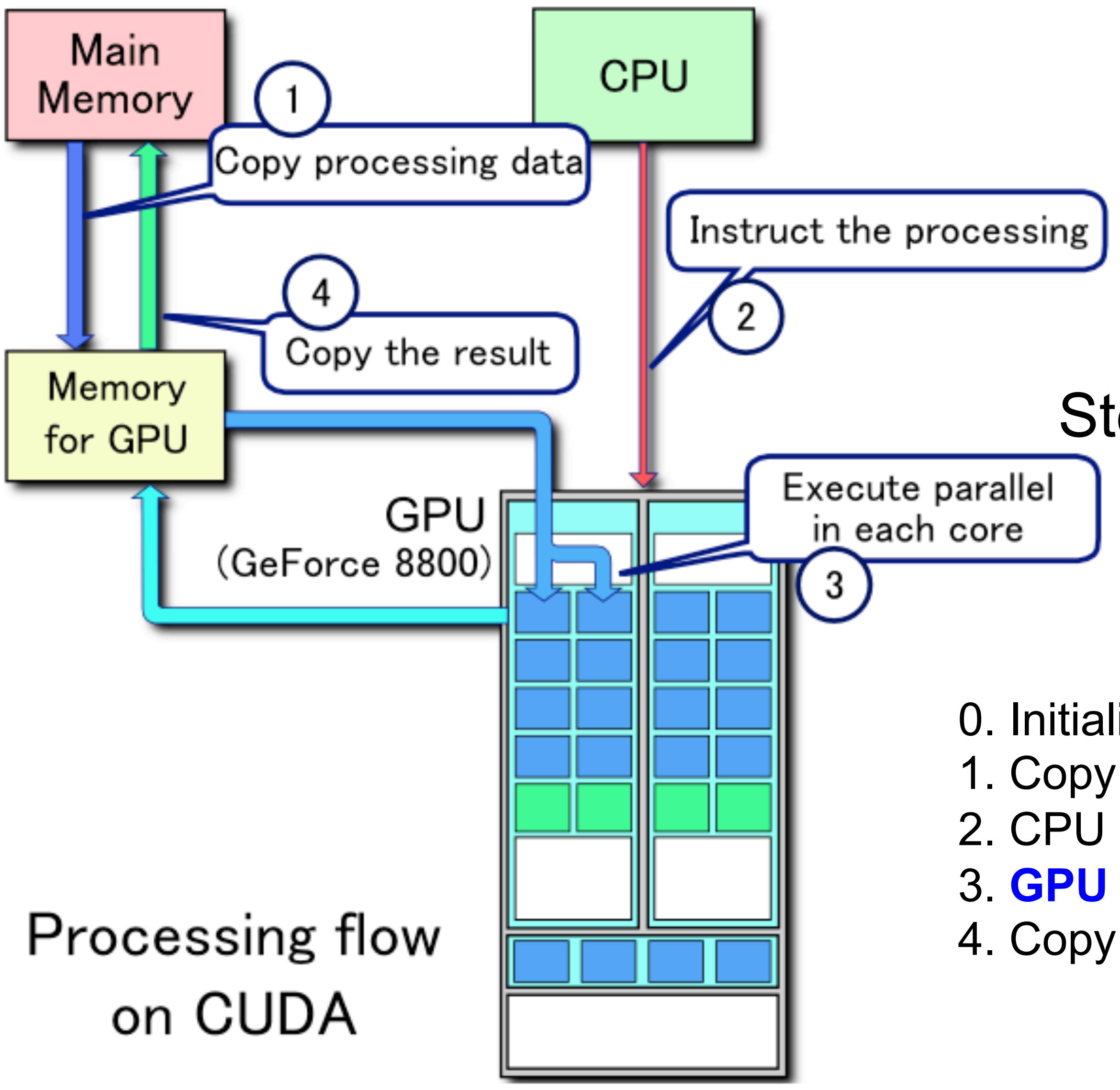
CUDA Software Environment

__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 :

```
__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

0. Initialize data (halos) on CPU
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

A Code Example

```
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
// h_C contains the result in 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);
```

```
// 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];
}
```

```
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 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 3Dlblock of size (Dx, Dy, Dz),

the thread ID of a thread of index (x, y, z) is

(x + y Dx + z Dx Dy).

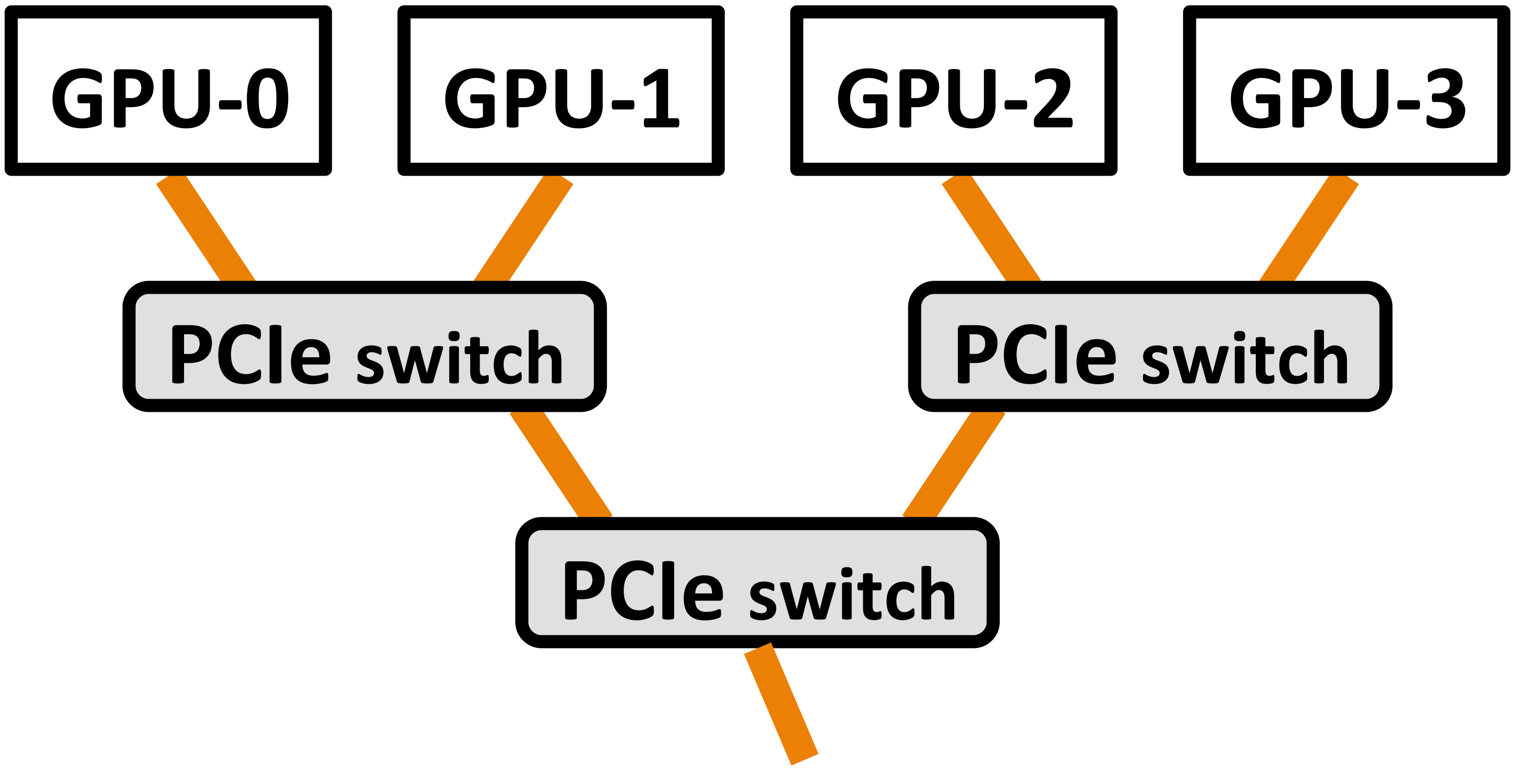
Device Thread Hierarchy Example 1

```
// 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 multipleGPUs 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 QF
- QPI and PCIe don't agree on P2P protocol
  - CUDA API will stage the data via host memory
-


GPU-0
GPU-1
GPU-2
GPU-3
PCIe switch
PCIe switch
PCIe switch

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 controlGPUs

Use of multiple GPUs on the same node

```
int devs =4

for (int d=0; d < devas; 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:
http://wiki.accelereyes.com/wiki/index.php/Function_List

How can Jacket help?

Partial support - Not every Matlab calculation can benefit

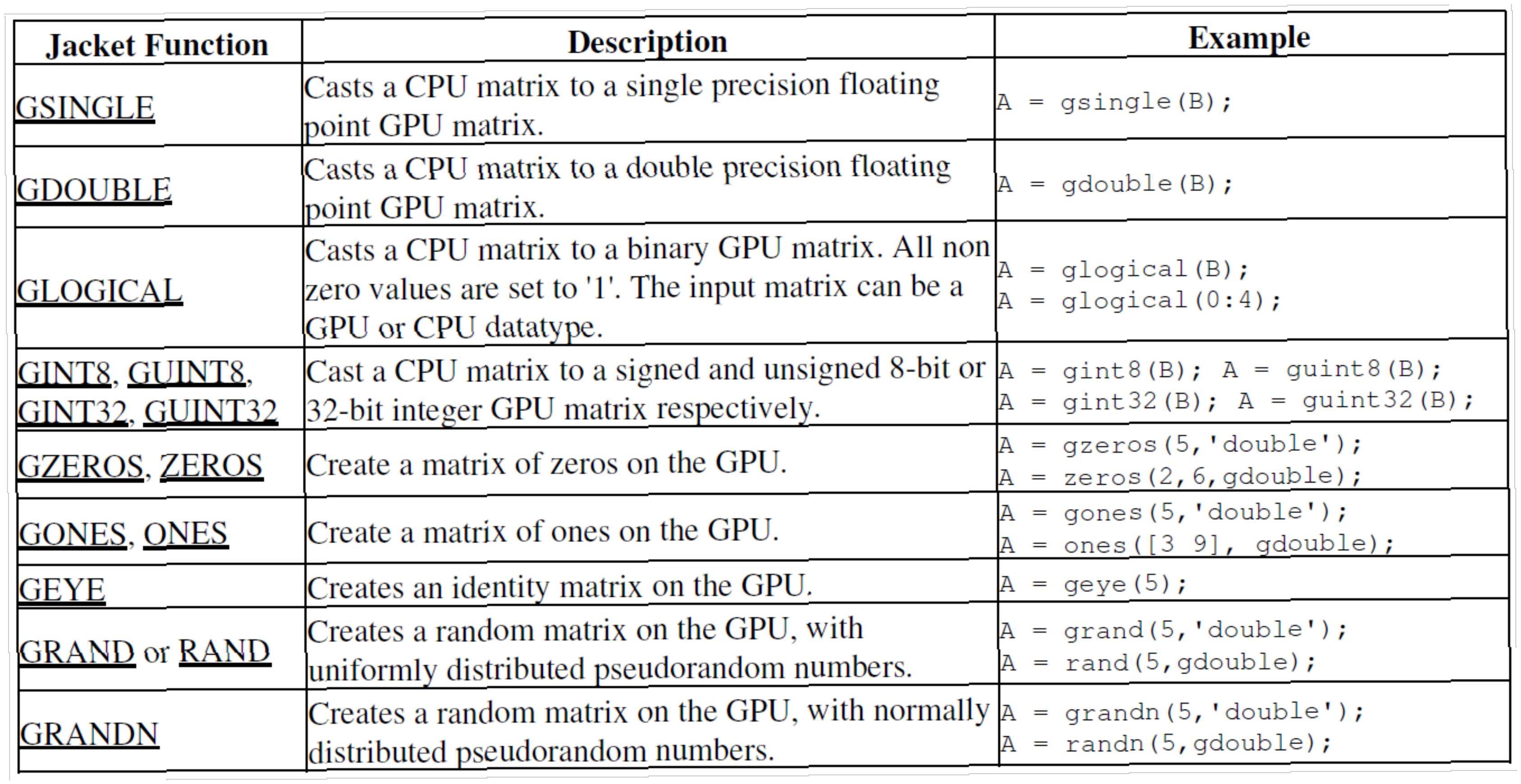
Hot spot – part of the code consumes most of the CPU time

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



Jacket Function	Description	Example
GSINGLE	Casts a CPU matrix to a single precision floating point GPU matrix.	`A = gsingle(B);`
GDOUBLE	Casts a CPU matrix to a double precision floating point GPU matrix.	`A = gdouble(B);`
GLOGICAL	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.	`A = glogical(B);` `A = glogical(0:4);`
GINT8, GUINT8, GINT32, GUINT32	Cast a CPU matrix to a signed and unsigned 8-bit or 32-bit integer GPU matrix respectively.	`A = gint8(B); A = guint8(B);` `A = gint32(B); A = guint32(B);`
GZEROS, ZEROS	Create a matrix of zeros on the GPU.	`A = gzeros(5,'double');` `A = zeros(2,6,gdouble);`
GONES, ONES	Create a matrix of ones on the GPU.	`A = gones(5,'double');` `A = ones([3 9], gdouble);`
GEYE	Creates an identity matrix on the GPU.	`A = geye(5);`
GRAND or RAND	Creates a random matrix on the GPU, with uniformly distributed pseudorandom numbers.	`A = grand(5,'double');` `A = rand(5,gdouble);`
GRANDN	Creates a random matrix on the GPU, with normally distributed pseudorandom numbers.	`A = grandn(5,'double');` `A = randn(5,gdouble);`

Basic functions

Jacket Function	Description	Example
GHELP	Retrieve information on the Jackt	`ghelp sum;`
GACTIVATE	Used for manual activation of a Jacket license.	`gactivate;`
GSELECT	Select or query which GPU is in use.	`gselect(0);`

GFOR	Executes FOR loop in parallel on GPU.	`gfor n = 1:10;`
		`% loop body`
		`gend;`

GCOMPILE	Compile M-code directly into a single CUDA kernel.	`my_fn =gcompile('filename.m')`

GPROFILE	Profile code to compare CPU versus GPU runtimes.	`gprofile on; foo; gprofile off;`
		`gprofile report;`

GPROFVIEW	Visual representation of profiling data.	`gprofview;`
GEVAL	Evaluate computation and leave results on GPU.	`geval;`
GSYNC	Block until all queued GPU computation is complete.	`gsync(A);`
GCACHE	Save GPU compiled code for given script.	`gcache;`
GLOAD	Load from disk directly into the GPU. Requires the Jacket SDK.	`gload('filename');`
GSAVE	Save data to disk as text file directly from the GPU. Requires the Jacket SDK.	`gsave('filename', A);`
GREAD	Load from disk directly into the GPU, with option to specify the byte range. Requires the Jacket SDK.	`gread('filename',`
`OFFSET, BYTES);`		
GWRITE S	Save data to disk directly from the GPU, with option to specify the byte range. Requires the Jacket SDK.	`gwrite('filename', OFFSET, DATA);`
Graphics	Library Functions contained in the Graphics Library.	`gplot(A);`

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