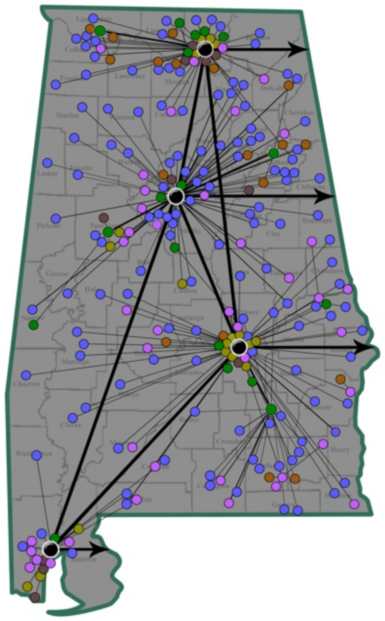
Introduction to GPU Programming with CUDA and OpenACC



Alabama Supercomputer Center
Alabama Research and Education Network

Contents

Topics

- Why GPU chips and CUDA?
- GPU chip architecture overview
- CUDA programming
- Queue system commands
- Other GPU programming options
- OpenACC programming
- Comparing GPUs to other processors

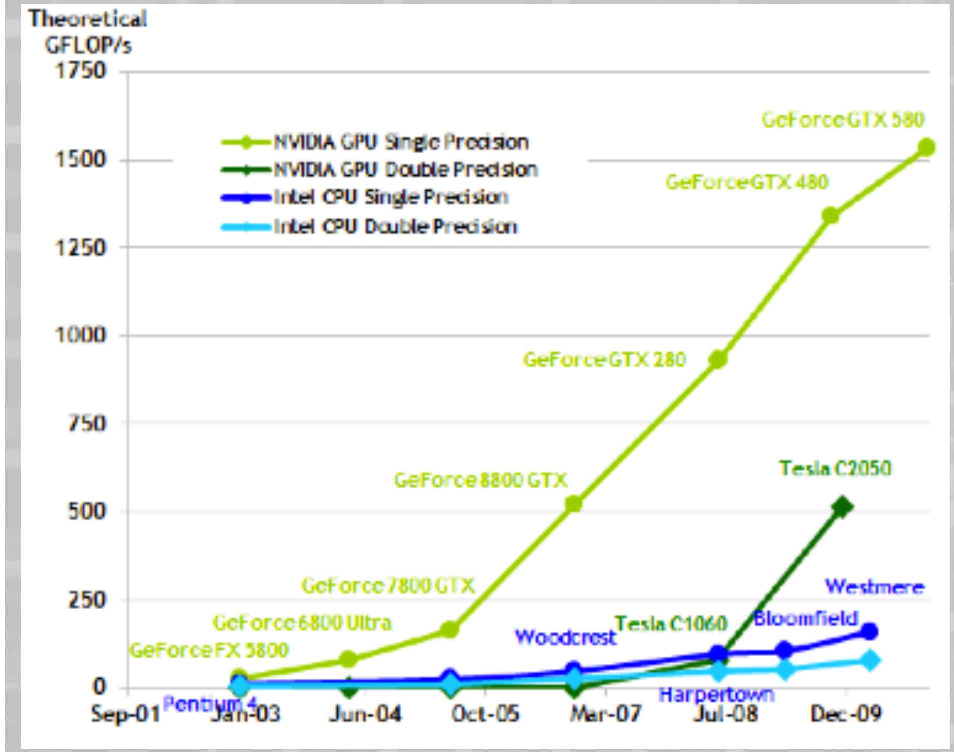
What is a GPU chip?

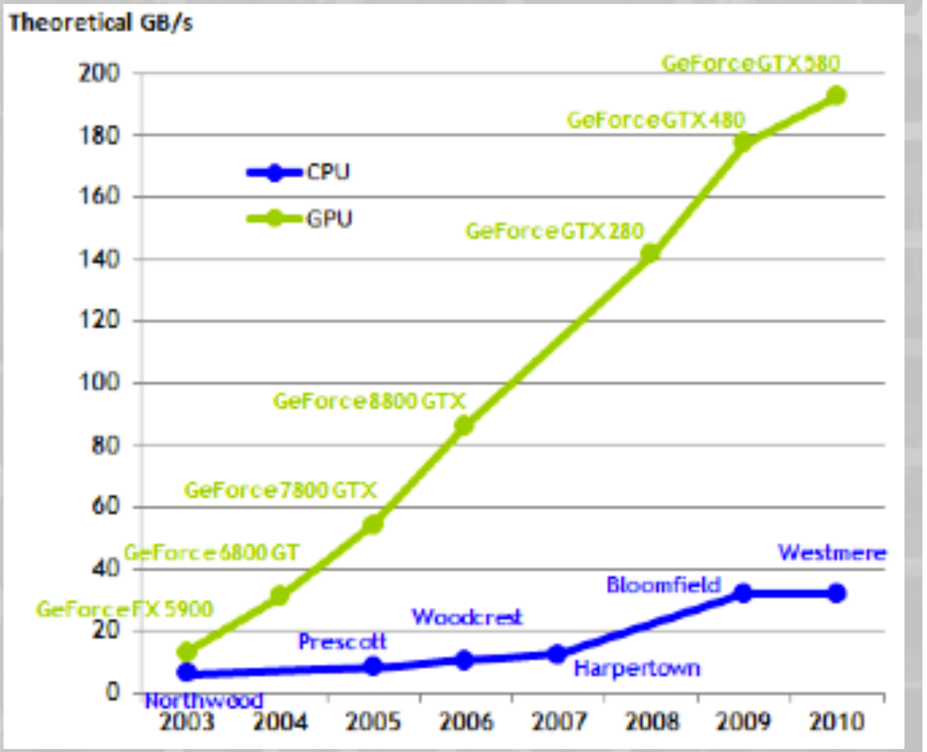
- A Graphic Processing Unit (GPU) chips is an adaptation of the technology in a video rendering chip to be used as a math coprocessor.

- The earliest graphic cards simply mapped memory bytes to screen pixels – i.e. the Apple][in 1980.
- The next generation of graphics cards (1990s) had 2D rendering capabilities for rendering lines and shaded areas.
- Graphics cards started accelerating 3D rendering with standards like OpenGL and DirectX in the early 2000s.
- The most recent graphics cards have programmable processors, so that game physics can be offloaded from the main processor to the GPU.
- A series of GPU chips sometimes called GPGPU (General Purpose GPU) have double precision capability so that they can be used as math coprocessors.

GPU

Why GPUs?





Comparison of peak theoretical GFLOPs and memory bandwidth for NVIDIA GPUs and Intel CPUs over the past few years.

Graphs from the NVIDIA CUDA C Programming Guide 4.0.

CUDA Programming Language

The GPU chips are massive multithreaded, manycore SIMD processors.

SIMD stands for Single Instruction Multiple Data.

Previously chips were programmed using standard graphics APIs (DirectX, OpenGL).

CUDA, an extension of C, is the most popular GPU programming language. CUDA can also be called from a C++ program.

The CUDA standard has no FORTRAN support, but Portland Group sells a third party CUDA FORTRAN.

Nvidia GPU Models

T10

- 30 multiprocessors with
  - 8 single precision thread processors
  - 2 special function units
  - Double precision unit
- 1.3 GHz
- 240 cores per chip
- 1036.8 GFLOP single
- 86.4 GFLOP double

Fermi (T20)

- 14 multiprocessors with
  - 32 thread processors are single & double add/multiply
  - 4 special function units
  - 2 clock ticks per double precision operation
- 1.15 GHz
- Faster memory bus
- Multiple kernels (subroutines) can run at once
- 448 cores per chip
- 1288 GFLOP single
- 515.2 GFLOP double

Kepler (K20)

- 13 multiprocessors with
  - 192 single precision thread processors
  - 64 double precision thread processors
  - 32 special function units
- 0.706 GHz
- Threads can spawn new threads (recursion)
- Multiple CPU cores can access simultaneously
- 2496 cores per chip
- 3520 GFLOP single
- 1170 GFLOP double

GPU Programming Example

CUDA

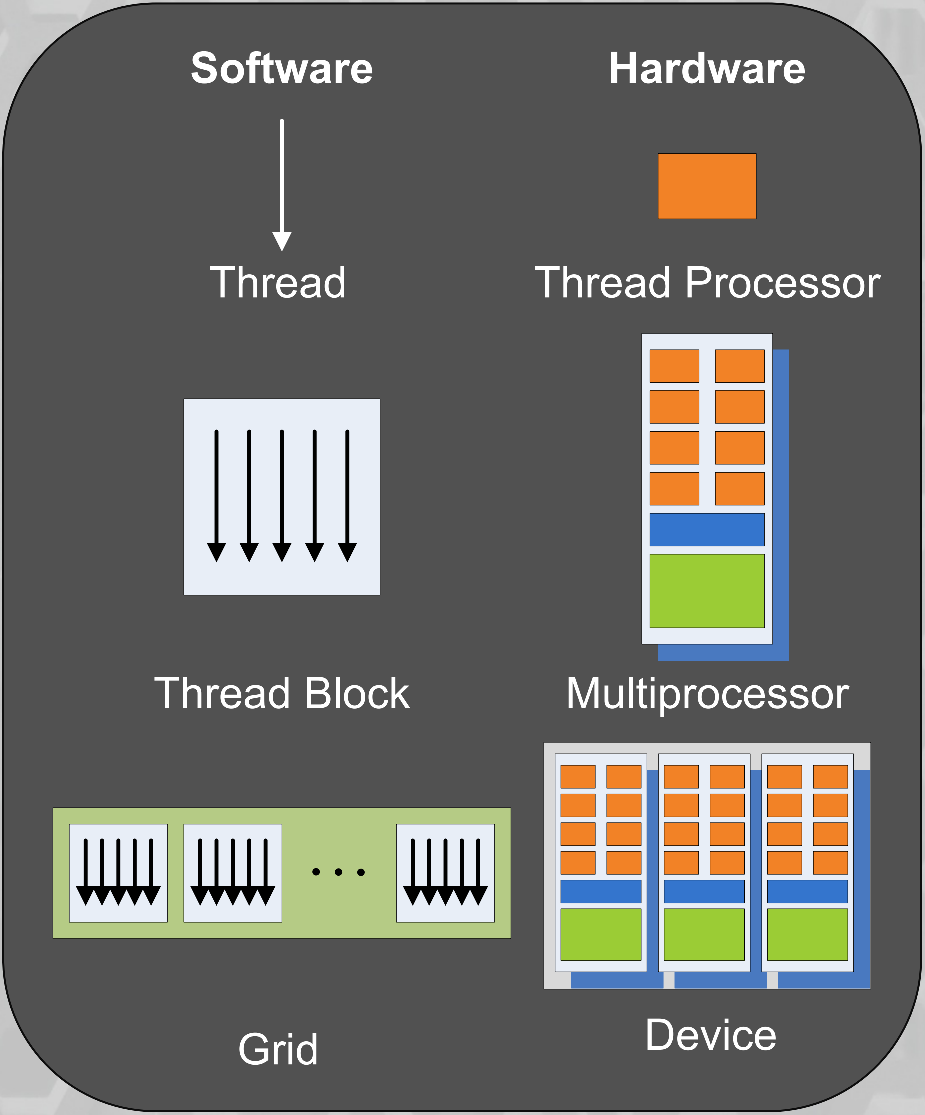
```
// CPU only matrix add
int main() {
 int i, j;
 for (i=0;i<N;i++) {
  for (j=0;j<N;j++) {
   C[i][j]=A[i][j]+B[i][j];
  }
 }
}
```

```
// GPU kernel
__global__ gpu(A[N][N], B[N]
    [N], C[N][N]) {
 int i = threadIdx.x;
 int j = threadIdx.y;
 C[i][j]=A[i][j]+B[i][j];
}

int main() {
 dim3 dimBlk(N,N);
 gpu<<1,dimBlk>>(A,B,C);
}
```

Code

GPU Execution Model



Thread is a single execution of a kernel, and all execute the same code

Threads within a block have access to shared memory for local cooperation

Kernel launched as a grid of independent thread blocks, and only a single kernel executes at a time (on T10)

SIMD Programming

CUDA

1. Copy an array of data to the GPU.
2. Call the GPU, specifying the dimensions of thread blocks and number of thread blocks (called a grid).
3. All processors are executing the same subroutine on a different element of the array.
4. The individual processors can choose different branch paths. However, there is a performance penalty as some wait while others are executing their branch path.
5. Copy an array of data back out to the CPU.

GPU programming is more closely tied to chip architecture than conventional languages.

Multiple types of memory help optimize performance

Code

Motherboard

Page locked host memory – This allows the GPU to see the memory on the motherboard. This is the slowest to access, but allows the GPU to access the largest memory space.

GPU chip

Global memory – Visible to all multiprocessors on the GPU chip.
Constant memory – Device memory that is read only to the thread processors and faster access than global memory.
Texture & Surface memory – Lower latency for reads to adjacent array elements.

Multiprocessor

Shared memory – Shared between thread processors on the same multiprocessor.

Thread processor

Local memory – accessible to the thread processor only. This is where local variables are stored.

Calling CUDA from C++

CUDA

- #include <cuda_runtime>
- The function call in file.cpp calls a function in file.cu which is

```
extern "C" void function();
```

- That function in turn calls a function that is

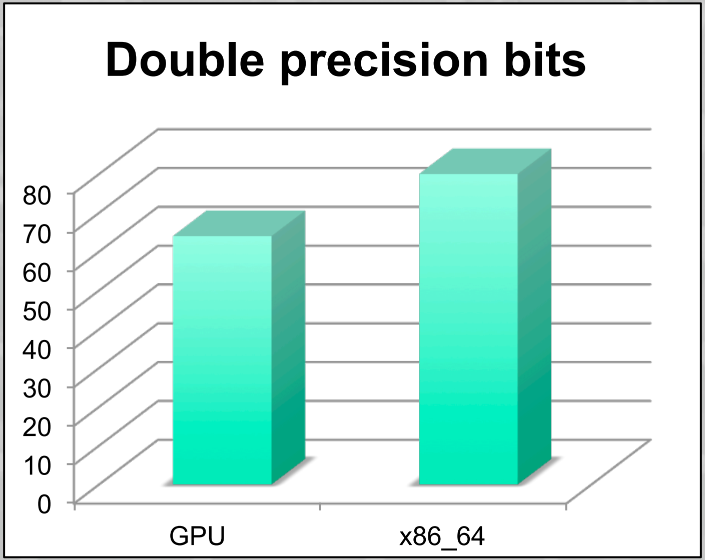
```
__global__ void function()
```

CAUTION: The C++ program must be named file.cpp (not file.cc). Files named with extension .cc can be erased by the make process.

Double Precision Support

CUDA

- Double precision on GPUs is true 64 bit. Double precision on x86 chips is 80 bit extended double precision.
- For double precision you must specify the architecture like this -arch=compute_13 -code=sm_13
- Use double precision variables in the Makefile like CUFILES_sm_13
- For double precision, do NOT use -use_fast_math



- The FLOPS ratings show that the Fermi chips should be about 6X the performance of T10 chips for double precision operations. However, the Fermi chips have an 8:1 ratio of thread processors to special function units, and the T10 chips have a 4:1 ratio. One of our tests that utilizes double precision and special functions showed a 2.5X improvement in speed in going from T10 to Fermi chips.

CUDA SDK Directory Tree

CUDA

- Unlike most compilers, CUDA is designed to work within a directory tree, rather than having source code, object and executable in the same directory. The common tools in that directory tree must be compiled before compiling your own programs.

- The source code goes in

```
~/CUDA_SDK_4.0/C/src/MYPROGRAM
```

- The object files get put in

```
~/CUDA_SDK_4.0/C/src/MYPROGRAM/obj/x86_64/release
```

- The executables get put in

```
~/CUDA_SDK_4.0/C/bin/linux/release
```

- The Makefile sets just a few variables, then loads a complex make process with the command

```
include ../../common/common.mk
```

Using nvcc outside the directory tree

CUDA

- Compiling from within the CUDA directory tree is not always desired.
- The use of the nvcc compiler directly (not with the provided Makefile) is supported in version 3.0 and 5.0, but not in version 4.0
- The available compile flags can be found with the command “`nvcc --help`”.
- An error free compile and link does not necessarily make a functioning executable.
- In order to find out how the default build process works, type the following

```
make clean
make -n
```

Changes between CUDA versions

CUDA

- CUDA is still evolving. Here are some of the things that have changed from version 3 to 4 to 5 to 6

- Makefile format
- Compile commands
- Mechanisms for error trapping.
- Header files
- nvcc switched from using GCC to using LLVM
- Processor support
- MPI integration
- Easier memory management called “unified memory”
- C++ 11 support
- Template support
- New GPU based math libraries

Catching Error Messages

CUDA

WARNING

By default, NO run time error messages are generated!

- **In order to generate error messages, the following steps should be followed.**

- **The .cu file should include**

```
#include "cutil_inline.h”
```

- **Allocate data with cutilSafeCall, like this**

```
cutilSafeCall (cudaMalloc( &Data, numx*numy*sizeof(double)));
```

- **Immediately after running a function on the GPU, call**

```
cutilCheckMsg(”MYFUNCTION<<<>>> failed\n");
```

Common Error Messages

CUDA

CAUTION: Error messages are not very indicative of the problem.

- **An error like this might mean you forgot to include the header file for CUDA**

```
myprogram.cc:81: error: expected constructor,
 destructor, or type conversion before 'void'
```

- **An error like this indicates you have exceeded a limit like the maximum number of threads per block.**

```
(9) invalid configuration argument
```

- **If you get an error saying that -lcuda can't be found, it means that the compile must be done on one of the nodes with GPU chips installed on it.**

Common Error Messages

CUDA

- Some things that you would expect to be compile time errors will show up as run time errors. For example, incorrectly passing arguments to functions.

- You can get this error because of a thread synchronization problem. Putting in cudaDeviceSynchronize() calls can fix the problem.

```
==31102== Error: Internal profiling error
1719:999
```

- If you are getting memory errors, try calling the program like this.

```
cuda-memcheck program [arguments]
```

What algorithms work well on GPUs

Code

- Doing the same calculation with many pieces of input data.
- The number of processing steps should be at least an order of magnitude greater than the number of pieces of input/output data.
- Single precision performance is better than double precision.
- Algorithms where most of the cores will follow the same branch paths most of the time.
- Algorithms that require little if any communication between threads.

Adoption of GPUs at the Alabama Supercomputer Center

Code

Good

- Recent versions of Amber perform well on GPUs and are being used for production work.
- Several universities have integrated GPU programming into the curriculum.
- About 5% of the applications at the Alabama Supercomputer Center have GPU versions.

Disappointing

- Early tests with BEAST, NAMD, and Quantum Espresso are less than exciting. Not all algorithms are converted to GPU.

Status

- The GPU offering remains a small test bed of 8 T10 chips, 8 Fermi chips, and 16 Kepler K20 chips.

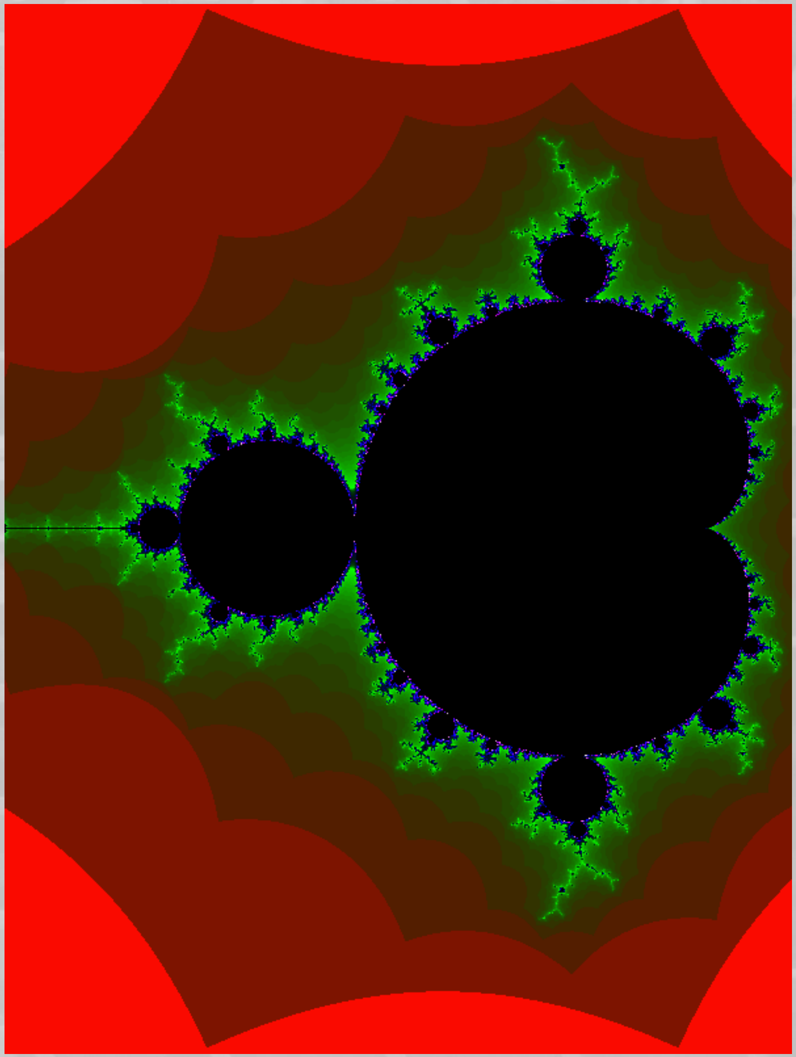
Performance Optimization

Code

- Utilize the type of memory that will give the best performance for the algorithm.
- The chip is made for zero latency swapping threads so that a different warp (group of usually 32 threads) can run while one warp is waiting on IO, SFU, DPU. Thus it is often best to have more threads than thread processors.
- The best number of threads/block depends on the program, but should be a multiple of 32 such as 64, 128, 192, 256, 768.
- The grid size should be at least the number of multiprocessors, and also works well as a multiple of the number of multiprocessors.
- If __syncthreads() slows the code, use more, smaller blocks.

Mandelbrot Test

- This is a single precision Mandelbrot diagram generator that is used as a simple parallel programming example.
- The large test run took 1 minute, 34 seconds on a single 2.26 GHz Nehalem processor.
- The same test took 9 seconds on a T10 GPU after minimal optimization of thread blocks.
- This is a 10x speed up, but not the 100x that marketing claims suggest is possible.
- In this case, the conditional do-while inner loop probably caused some cores to sit idle waiting for the rest to reach their break points.



Validation of Results

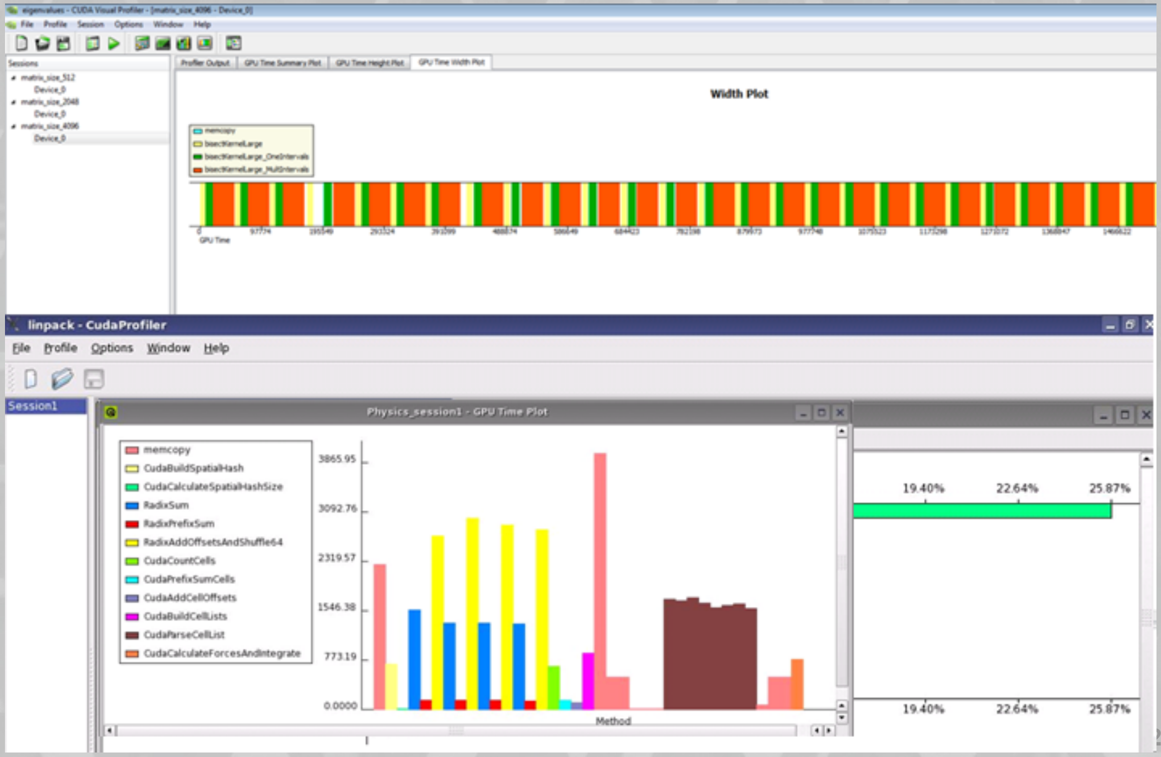
Code

- Validation is usually done using a gold kernel and maybe a silver kernel.
- Gold Kernel - data processed on the CPU with carefully checked output. You compare the CUDA output to the gold output to make sure the numerical accuracy is within acceptable limits.
- Silver Kernel - data processed on the GPU without optimization or algorithmic enhancements. This is the first step in GPU implementation. Again, comparing optimized kernel to silver kernel shows if the optimization reduced accuracy.
- Both of these usually use the simplest, most naive version of the algorithm (i.e. rectangle rule integration).

Other CUDA Tools

CUDA

- CUDA Memory Checker (cuda-memcheck) can be used to find memory violations
- CUDA debugger (cuda-gdb) is an extension of the GNU debugger for Linux
- NVIDIA Parallel Nsight is a debugger for Microsoft Visual Studio
- CUDA Visual Profiler

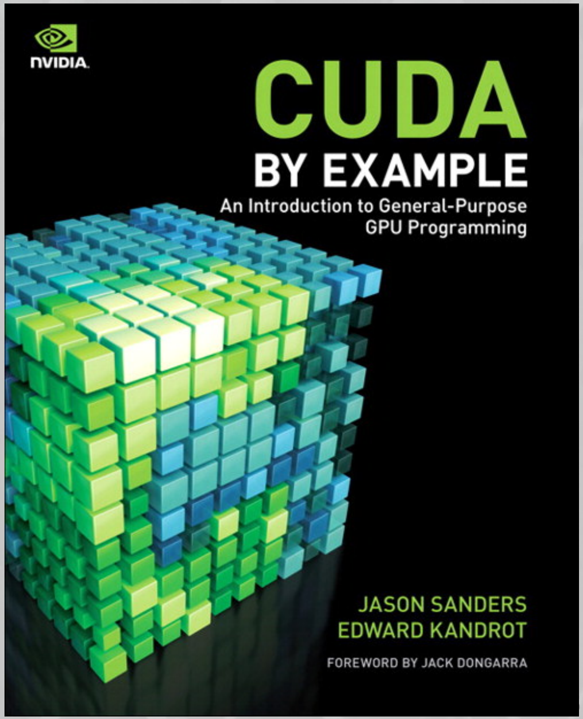


CUDA References

Doc

- **On the Alabama Supercomputer Center systems, documentation is in the directory /opt/asn/doc/gpu**
  - Start with README.txt and TIPS.txt
  - CUDA_C_Getting_Started_Linux.pdf
  - CUDA_C_Programming_Guide.pdf
  - CUDA_C_Best_Practices_Guide.pdf
  - Examples are in the portland_accelerator and portland_cuda_fortran directories
  - There is more information in the supplmental_docs directory

- **A good introduction to CUDA programming**
  - "CUDA BY EXAMPLE" by J. Sanders, E. Kandrot, Addison Wesley, 2011.


NVIDIA
CUDA
BY EXAMPLE
An Introduction to General-Purpose GPU Programming
JASON SANDERS
EDWARD KANDROT
FOREWORD BY JACK DONGARRA

GPUs & the Queue System

- The queue system at the Alabama Supercomputer Center has a couple commands for submitting work to the queues.
- The “`gpu_interactive`” command opens an interactive session on a GPU node. This should be used for compiling, only if it will not compile on the login node.
- The “`run_gpu`” command is used for submitting all production work to the queue.
- Only one GPU is available to a job. This is a policy restriction due to the limited number of GPU chips available.

Code

Other GPU Programming Options

- PGI Accelerator is a commercial compiler that allows programming NVIDIA GPUs with OpenACC, a syntax similar to OpenMP.
- OpenMP is starting to release GPU features.
- OpenCL – is a language under development for parallel programming of many different hardware architectures with a common syntax.
- There are CUDA plugins for Python, Matlab, and Mathematica
- Math Libraries
  - cuSOLVER (BLAS, Lapack)
  - cuFFT
  - NVIDIA Performance Primitives library – NPP
  - GPULib
  - FLAGON – Fortran-9x library
  - Thrust (C++11)
- Several more came and went already

OpenACC Example

OpenACC

```
// OpenACC matrix add
int main() {
 int i, j;
#pragma acc kernels loop gang(32), vector (16)
 for (i=0;i<N;i++) {
#pragma acc loop gang(16), vector(32)
  for (j=0;j<N;j++) {
   C[i][j]=A[i][j]+B[i][j];
  }
 }
}
```

- openACC is easier to program than CUDA
- but less efficient, so the program wont run as fast

Common OpenACC directives

OpenACC

- OpenACC directives in C and C++

  #pragma acc DIRECTIVE
- OpenACC directives in Fortran

  !$acc DIRECTIVE

  lines of Fortran code

  !$acc end DIRECTIVE
- Directive to attempt automatic parallelization

  #pragma acc kernels
- Directive to parallelize the next loop

  #pragma acc parallel loop
- Directive to specify which variables are copied, and which are local

  #pragma acc data copy(A), create(Anew)

The data directive is often needed to cut out data bottlenecks

Compiling and Running

- Typical compile command for C

```
pgcc -acc -Minfo=accel -ta=nvidia -o file file.c
```

- Environment variable to print GPU use information at run time

```
export PGI_ACC_TIME=1
```

- The program runs slightly slower with this turned on

- Environment variable to print out information about data transfers to the GPU at run time

```
export PGI_ACC_NOTIFY=3
```

- This slows down execution significantly

Ideal cases for OpenACC

OpenACC

- Programs where one or a few small sections of the program are responsible for most of the CPU time.

- Loops with many iterations.
- Loops with no data dependencies between iterations.
- Loops that work on many elements of large arrays.
- Loops where functions can be inlined.

- Conditional statements are OK, but better if you can guess in advance which batches of data will follow the same branch.

- Portland Group compilers create programs with code for three generations of GPUs; Tesla, Fermi, & Kepler

What Does NOT work well

OpenACC

- Loops with IO statements.
- Loops with early exits, including do-while loops.
- Loops with many branches to other functions.
- Pointer arithmetic

Confusingly, a failed compile creates a single processor executable.

Code

OpenACC vs. CUDA

- CUDA creates software for nVidia GPUs only. OpenACC can program GPUs, Opteron, ATI, APUs, Xeon, and Xeon Phi.
- OpenACC does loop level parallelization. CUDA parallelizes at the subroutine level.
- OpenACC is easier to program, or adapt an existing code.
- CUDA is currently used more widely.
- Some algorithms can be implemented in CUDA, but not in OpenACC. i.e. recursion or early exit loops
- OpenACC is newer (version 2.0 is out). CUDA is on version 7
- Both are still undergoing significant changes.

- CUDA programs usually run faster (perhaps 30%).

OpenACC documentation

Doc

- Look at the Getting Started documentation and videos at openacc-standard.org
- https://developer.nvidia.com/content/openacc-example-part-1
- The PGI Acclerator Compilers OpenACC Getting Started Guide

  http://www.pgroup.com/doc/openACC_gs.pdf
- There are example programs in the directories

  /opt/asn/doc/pgi/accelerator_examples

  /opt/asn/doc/pgi/openacc_example
- There are tips for best results in the file

  /opt/asn/doc/gpu/openacc_tips.txt
- OpenACC 2.0 examples are at

  http://devblogs.nvidia.com/parallelforall/7-powerful-new-features-openacc-2-0/

Unfortunately, once you get past the introductory documentation, you will need to read the OpenACC technical specifications and ask questions on user forums to maximize performance with OpenACC.

Comparing GPUs to other types of processors

Vector/SIMD extensions

SSE

- 4 x86 ops to add two single precision, four-component vectors

```
vector_result.x = vector_1.x + vector_2.x;
vector_result.y = vector_1.y + vector_2.y;
vector_result.z = vector_1.z + vector_2.z;
vector_result.w = vector_1.w + vector_2.w;
```

- Using 128-bit SSE registers, pack vector components into a single register per vector to reduce this from 4 scalar addition ops to a single SSE vector addition

- Intel's Sandy Bridge architecture (used in UV) introduced AVX instructions that further widens vector data path from 128 to 256 bits, potentially resulting in up to a 2x performance improvement for some applications

FLOPS vs Chip Architecture

Chips

- The FLOPS (FLoating point Operations Per Second) rating is NOT a good comparison of GPU performance relative to conventional processor performance.
  - FLOPS rating is usually a poor way to compare any types of chips.

- The FLOPS rating for conventional processors includes the vector math circuitry for SSE instructions. If your program cannot use SSE instructions, a conventional processor may under-perform it's FLOPS rating, and the GPU may approach the GPU FLOPS performance.

- If your program has significant communication between threads, or different threads take different branch paths, the GPU may do worse than the FLOPS ratings suggest.

GPU vs. Xeon vs. Xeon Phi

Chips

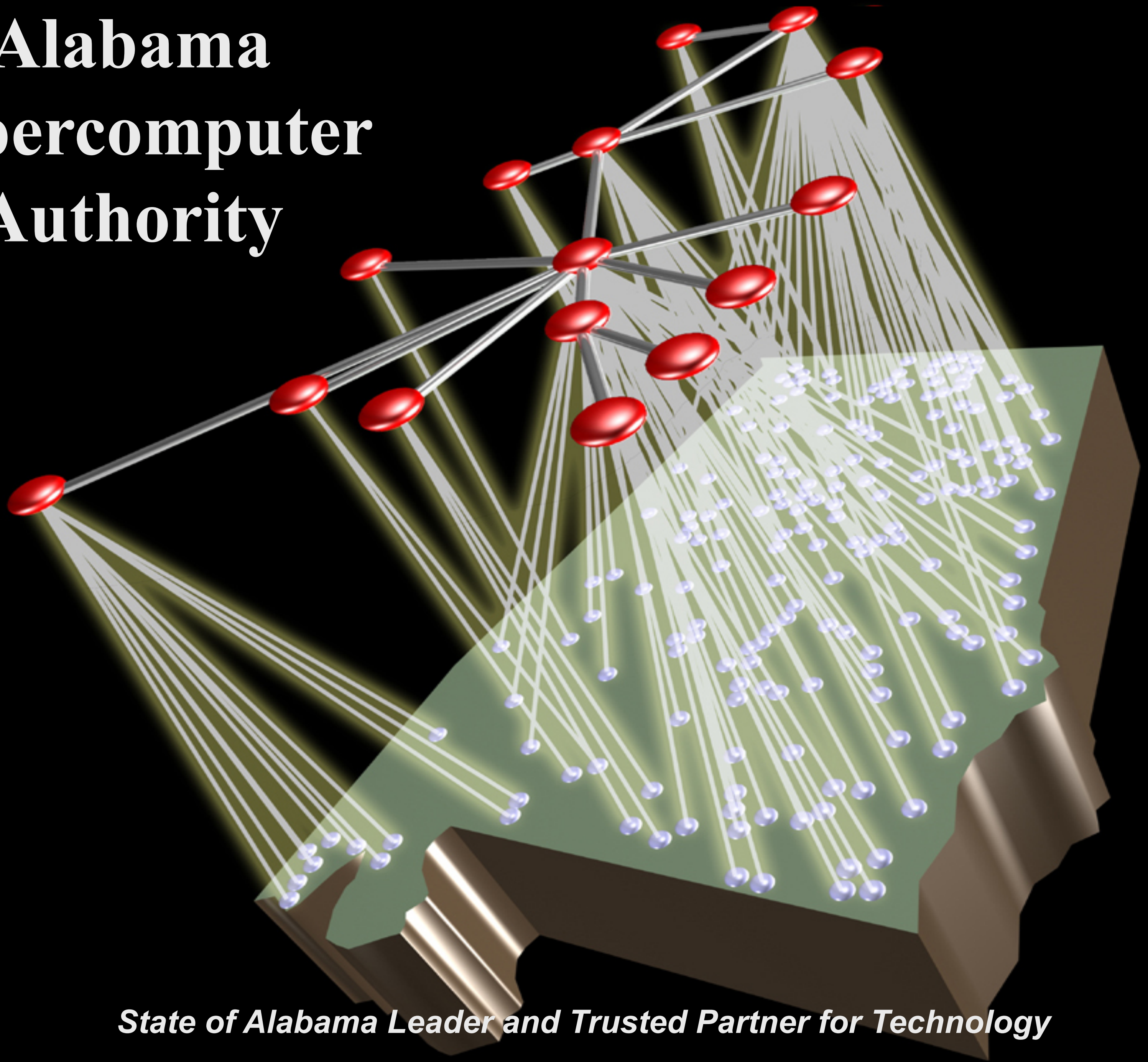
- Xeon Phi is a processor with 57-61 x86 compatible cores running at 1.053 to 1.238 GHz.
- Xeon Phi is NOT a chip with a bunch of Xeon processor cores. The cores on Phi are less powerful (about 1/5 speed).
- Xeon Phi is a new chip architecture called MIC (Many Integrated Cores). The next MIC chip will be Knights Landing.
- OpenMP parallelized software will run on Xeon Phi, but runs faster if you do some work to manage memory access bottle necks.
- Xeon Phi has SSE vector mathematics instructions. GPUs do not do vector math.

Summary

Done

- There is a lot of interest in the HPC community about using GPU chips because GPUs can give 10-300 fold the processing capacity for the dollar spent on hardware... provided you have invested the effort to port the software to that architecture.
- GPUs are easier to program than other coprocessor technologies (i.e. FPGAs).
- The GPGPU programming market is currently dominated by Nvidia chips and the CUDA programming language.
- CUDA is the most mature of the GPU programming options, but still an early stage technology.
- OpenACC is increasing in popularity.
- CUDA is more closely tied to hardware than higher level languages like C++.
- Many experts predict that OpenCL could become the preferred GPU programming method if future versions achieve the intended goal of being a “write once – run anywhere” parallel language.

Alabama Supercomputer Authority



State of Alabama Leader and Trusted Partner for Technology