Introduction to the CUDA Toolkit as an Application Build Tool

Adam DeConinck
HPC Systems Engineer, NVIDIA
What this talk will cover:

The CUDA 5 Toolkit as a toolchain for HPC applications, focused on the needs of **sysadmins** and **application packagers**

- Review GPU Computing concepts
- Example application build workflows
- Important features of CUDA C compilation process
- Other CUDA compilers
- Common libraries
- A few changes with CUDA 5.5

What this talk won’t cover:

- Developing software for GPUs
- General sysadmin of a GPU cluster
- Earlier versions of CUDA (mostly)
- Anything to do with Windows
CPU vs GPU

Latency Processor + Throughput processor

CPU

GPU
Low Latency or High Throughput?

**CPU**
- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

**GPU**
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation
Processing Flow

1. Copy input data from CPU memory to GPU memory
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory
Anatomy of a CUDA Application

- **Serial** code executes in a **Host** (CPU) thread
- **Parallel** code executes in many **Device** (GPU) threads across multiple processing elements
3 Ways to Accelerate Applications

**Applications**

- **Libraries**
  - “Drop-in” Acceleration

- **OpenACC Directives**
  - Compiler directives (Like OpenMP)

- **Programming Languages**
  - Most common: CUDA C
  - Also CUDA Fortran, PyCUDA, Matlab, ...
3 Ways to Accelerate Applications

Applications

- Most of the talk will focus on CUDA Toolkit (CUDA C)
- Will touch on OpenACC and common libraries at the end of the talk

Programming Languages

- Most common: CUDA C
- Also CUDA Fortran, PyCUDA, Matlab, ...
Working with the CUDA Toolkit
CUDA Toolkit

- Free developer tools for building applications with CUDA C/C++ and the CUDA Runtime API

- Includes (on Linux):
  - nvcc compiler
  - Debugging and profiling tools
  - Nsight Eclipse Edition IDE
  - NVIDIA Visual Profiler
  - A collection of libraries (CUBLAS, CUFFT, Thrust, etc)

- Currently the most common tool for building NVIDIA GPU applications
CUDA Toolkit environment module

#%Module
module-whatis "CUDA Toolkit 5.0"
set root /opt/cuda-5.0
set CUDA_HOME $root
prepend-path PATH $root/bin
prepend-path PATH $root/open64/bin
prepend-path CPATH $root/include
prepend-path LD_LIBRARY_PATH $root/lib64
Building a CUDA app

- CUDA doesn’t impose any specific build process, so most common build processes are represented in apps
  - configure/make/make install
  - cmake/make/make install
  - etc

- Similar workflow to MPI: point to nvcc correctly in your Makefile (like pointing to the right mpicc)
  - But you **always** have to use the “wrapper” compiler; not just a wrapper like mpicc to command-line options

- If CUDA support is optional, there’s often a configure option or macro to enable/disable it
  - --enable-cuda ... --with-cuda ... --enable-nvidia ... -DCUDA_ENABLE=1 ... 
  - No convention on what this option should be
Where’s CUDA?

Common to install CUDA somewhere other than /usr/local/cuda (i.e. clusters, environment modules), so build process often includes specifying a path

- **Common**: specify location of the CUDA toolkit using an environment variable
  - No convention on the name of this variable, though
  - CUDA_HOME=... is common
  - Also CUDA=, CUDA_PATH=, NVIDIA_CUDA=, ...
- OR a command line argument: `--with-cuda-lib=/opt/cuda` ...
- OR just hard-code /usr/local/cuda in the Makefile
  - I see this far too frequently.
NVCC Compiler

- Compiler for CUDA C/C++
- Uses the CUDA Runtime API
  - Resulting binaries link to CUDA Runtime library, libcudart.so
- Takes a mix of host code and device code as input
  - Uses g++ for host code
- Builds code for CPU and GPU architectures
- Generates a binary which combines both types of code
## Common NVCC Options

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Command-line flag</th>
<th>Equivalent for gcc</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>INCLUDES</td>
<td>--include-path -I</td>
<td>CPATH -I</td>
<td>Define additional include paths</td>
</tr>
<tr>
<td>LIBRARIES</td>
<td>--library-path -L</td>
<td>LD_LIBRARY_PATH -L</td>
<td>Define additional library paths</td>
</tr>
<tr>
<td></td>
<td>--optimize -O</td>
<td>-O</td>
<td>Optimization level for host code</td>
</tr>
<tr>
<td></td>
<td>-use_fast_math</td>
<td></td>
<td>Apply all device-level math optimizations</td>
</tr>
<tr>
<td>PTXAS_FLAGS</td>
<td>-Xptxas=-v</td>
<td></td>
<td>Print GPU resources (shared memory, registers) used per kernel</td>
</tr>
</tbody>
</table>
Most major MPIs now support addressing CUDA device memory directly
  - Do MPI_Send/MPI_Receive with pointers to device memory; skip cudaMemcpy step in application code

GPUDirect: do direct device-to-device transfers (skipping host memory)

OpenMPI, mvapich2, Platform MPI, … See NVIDIA DevZone for a full list
Support typically has to be included at compile time
Example: matrixMul

- Single CUDA source file containing host and device code
- Single compiler command using nvcc

$ nvcc -m64 -I../common/inc matrixMul.cu
$ ./a.out

[Matrix Multiply Using CUDA] - Starting...
GPU Device 0: "Tesla M2070" with compute capability 2.0
MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...done
...
Example: simpleMPI

- Simple example combining CUDA with MPI
  - Split and scatter an array of random numbers, do computation on GPUs, reduce on host node

- MPI and CUDA code separated into different source files, simpleMPI.cpp and simpleMPI.cu

- Works exactly like any other multi-file C++ build
  - Build the CUDA object file, build the C++ object, link them together
$ make

nvcc -m64 -gencode arch=compute_10,code=sm_10 -gencode arch=compute_20,code=sm_20 -gencode arch=compute_30,code=sm_30 -o simpleMPI.o -c simpleMPI.cu

mpicxx -m64 -o main.o -c simpleMPI.cpp

mpicxx -m64 -o simpleMPI simpleMPI.o main.o -L$CUDA/lib64 -lcudart
$ make

nvcc -m64 -gencode arch=compute_10,code=sm_10 -gencode arch=compute_20,code=sm_20 -gencode arch=compute_30,code=sm_30 -o simpleMPI.o -c simpleMPI.cu

(we’ll explain the -gencode bits later)

mpicxx -m64 -o simpleMPI simpleMPI.o main.o -L$CUDA/lib64 -lcudart
Example: OpenMPI

- Popular MPI implementation

- Includes CUDA support for sending/receiving CUDA device pointers directly, without explicitly staging through host memory
  - Either does implicit cudaMemcpy calls, or does direct transfers if GPUDirect support

- Configure options:
  --with-cuda=$CUDA_HOME
  --with-cuda-libdir=/usr/lib64 (or wherever libcuda.so is)
Example: GROMACS

- Popular molecular dynamics application with CUDA support (mostly simulating biomolecules)

- Version 4.5: CUDA support via OpenMM library, only single-GPU support

- Version 4.6: CUDA supported directly, multi-GPU support

- Requires Compute Capability \(\geq 2.0\) (Fermi or Kepler)
Example: GROMACS


tar xzf gromacs-4.6.tar.gz

mkdir gromacs-build

module load cmake cuda gcc/4.6.3 fftw openmpi

CC=mpicc CXX=mpiCC cmake ./gromacs-4.6 -DGMX_OPENMP=ON
-DGMX_GPU=ON -DGMX_MPI=ON -DGMX_PREFER_STATIC_LIBS=ON -D
DCMAKE_BUILD_TYPE=Release -DNCMAKE_INSTALL_PREFIX=./gromacs-build

make install
Example: GROMACS (cmake)

cmake defines a number of environment variables for controlling nvcc compiler

<table>
<thead>
<tr>
<th>Environment variables</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_HOST_COMPILER</td>
<td>Specify which host-code compiler to use (i.e. which gcc)</td>
</tr>
<tr>
<td>CUDA_HOST_COMPILER_OPTIONS</td>
<td>Options passed to the host compiler</td>
</tr>
<tr>
<td>CUDA_NVCC_FLAGS</td>
<td>Options passed to nvcc</td>
</tr>
</tbody>
</table>

GROMACS default value for CUDA_NVCC_FLAGS:
- gencode;arch=compute_20,code=sm_20;-gencode;arch=compute_20,code=sm_21;
- gencode;arch=compute_30,code=sm_30;
- gencode;arch=compute_30,code=compute_30;-use_fast_math;
CUDA C Build Process
What actually gets built by nvcc?

- NVCC generates **three** types of code:
  - Host object code (compiled with g++)
  - Device object code
  - Device assembly code (PTX)

- Compiler produces a “fat binary” which includes all three types of code

- Breaking changes in both NVIDIA object code and in PTX assembly can occur with each new GPU release

- PTX is forward-compatible, object code is not
Fat binaries

When a CUDA “fat binary” is run on a given GPU, a few different things can happen:

- If the fat binary includes object code compiled for the device architecture, that code is run directly.

- If the fat binary includes PTX assembly which the GPU understands, that code is Just-In-Time compiled and run on the GPU. (results in startup lag due to JIT compilation)

- If neither version are compatible with the GPU, the application doesn’t run.
Why do we care?

- A given CUDA binary is not guaranteed to run on an arbitrary GPU
- `nvcc` defaults to compiling for maximum compatibility (Compute Capability 1.0), so build options required for support of new features, better performance
- And if it does run, not guaranteed to get best performance
  - JIT compilation imposes a minor startup penalty
  - Your GPU may support newer PTX or object code features than are compiled in
- Mix of hardware you have in your cluster determines what options to include in your fat binaries
NVCC Build Process (simplified)

Input Files ➔ nvcc ➔ Device code ➔ nvopencc ➔ ptxas ➔ PTX (device assembly) ➔ fatbinary ➔ PTX and/or CUBIN ➔ gcc ➔ Host object code ➔ Combined object code

Host code ➔ gcc ➔ Host object code ➔ Combined object code
## Options to different stages

<table>
<thead>
<tr>
<th>Environment variables</th>
<th>Command-line options</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Xcompiler</td>
<td>Pass options directly to the (host) compiler/preprocessor (i.e. gcc)</td>
<td></td>
</tr>
<tr>
<td>-Xlinker</td>
<td>Pass options directly to the linker</td>
<td></td>
</tr>
<tr>
<td>-Xcudafe</td>
<td>Pass options directly to cudafe (pre-processor/splitter)</td>
<td></td>
</tr>
<tr>
<td>OPENCC_FLAGS</td>
<td>-Xopencc</td>
<td>Pass options directly to nvopencc, typically for steering device code optimization</td>
</tr>
<tr>
<td>PTXAS_FLAGS</td>
<td>-Xptxas</td>
<td>Pass options directly to the ptx optimizing compiler</td>
</tr>
</tbody>
</table>
Compute capability and device architecture

**Compute Capability**
- Defines the *computing features* supported by a given GPU generation
- Language features (i.e. double precision floats, various functions)
- Device features (size of shared memory, max thread block size, etc)
- Newer GPUs can run older PTX assembly code.

**GPU Architecture**
- Binary code is architecture-specific, and changes with each GPU generation
- Version of the object code.
- Different architectures use different optimizations, etc.
- Binary code from one architecture can’t run on another
NVCC Build Process (simplified)

nvcc Input Files

Host code

orc

gcc

nvopencc generates PTX assembly according to the *compute capability*

ptxas generates device binaries according to the *device architecture*

fatbinary packages them together

Device code

PTX (device assembly)

NVCC Build Process (simplified)

nvcc Input Files

Host code

orc

gcc

nvopencc generates PTX assembly according to the *compute capability*

ptxas generates device binaries according to the *device architecture*

fatbinary packages them together

Device code

PTX (device assembly)

NVCC Build Process (simplified)

nvcc Input Files

Host code

orc

gcc

nvopencc generates PTX assembly according to the *compute capability*

ptxas generates device binaries according to the *device architecture*

fatbinary packages them together

Device code

PTX (device assembly)

NVCC Build Process (simplified)

nvcc Input Files

Host code

orc

gcc

nvopencc generates PTX assembly according to the *compute capability*

ptxas generates device binaries according to the *device architecture*

fatbinary packages them together

Device code

PTX (device assembly)

NVCC Build Process (simplified)

nvcc Input Files

Host code

orc

gcc

nvopencc generates PTX assembly according to the *compute capability*

ptxas generates device binaries according to the *device architecture*

fatbinary packages them together

Device code

PTX (device assembly)

NVCC Build Process (simplified)

nvcc Input Files

Host code

orc

gcc

nvopencc generates PTX assembly according to the *compute capability*

ptxas generates device binaries according to the *device architecture*

fatbinary packages them together

Device code

PTX (device assembly)

NVCC Build Process (simplified)

nvcc Input Files

Host code

orc

gcc

nvopencc generates PTX assembly according to the *compute capability*

ptxas generates device binaries according to the *device architecture*

fatbinary packages them together

Device code

PTX (device assembly)

NVCC Build Process (simplified)

nvcc Input Files

Host code

orc

gcc

nvopencc generates PTX assembly according to the *compute capability*

ptxas generates device binaries according to the *device architecture*

fatbinary packages them together

Device code

PTX (device assembly)

NVCC Build Process (simplified)

nvcc Input Files

Host code

orc

gcc

nvopencc generates PTX assembly according to the *compute capability*

ptxas generates device binaries according to the *device architecture*

fatbinary packages them together

Device code

PTX (device assembly)

NVCC Build Process (simplified)

nvcc Input Files

Host code

orc

gcc

nvopencc generates PTX assembly according to the *compute capability*

ptxas generates device binaries according to the *device architecture*

fatbinary packages them together

Device code

PTX (device assembly)
Compute capability and device architecture

- When you compile code with NVCC, you can specify
  - Compute capabilities, which describe version of CUDA language & PTX. Compute capability named as `compute_XX`. (XX=version number)
  - Device architectures, which describe version of CUDA object code. Device architecture named as `sm_XX`.

- You can generate multiple versions of both the PTX and the object code to be included.

```
nvcc -m64 -gencode arch=compute_10,code=sm_10 -gencode arch=compute_20,code=sm_20 -gencode arch=compute_30,code=sm_30 -o simpleMPI.o -c simpleMPI.cu
```
GROMACS revisited

Default flags in GROMACS: CUDA_NVCC_FLAGS=
-gencode;arch=compute_20,code=sm_20;-
gencode;arch=compute_20,code=sm_21;-
gencode;arch=compute_30,code=sm_30;-use_fast_math;

Generates code for compute versions 2.0 (Tesla M2050/M2070), compute version 2.1 (Quadro 600, various GeForce) and 3.0 (Tesla K10)

To generate optimized code for Tesla K20, you’d add compute capability 3.5: -gencode arch=compute_35,code=sm_35
Common build strategies

“Lowest common denominator”
- I can get away with Compute Capability 1.3, so that’s what I’ll use
- --gencode arch=compute_13,code=sm_13
- Newer GPUs must JIT from PTX code

“Everything under the sun!”
- Compile for everything released when I wrote the Makefile
- --gencode arch=compute_10,code=sm_10 --gencode arch=compute_13,code=sm_13
- --gencode arch=compute_20,code=sm_20 --gencode arch=compute_30,code=sm_30
- --gencode arch=compute_35,code=sm_35

“Newest features only”
- Target the GPU I just bought, ignore earlier ones
- --gencode arch=compute_30,code=sm_30
## Command line options for specifying arch

<table>
<thead>
<tr>
<th>Long option</th>
<th>Short option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--gpu-architecture &lt;arch&gt;</code></td>
<td>-arch</td>
<td>Specify the GPU architecture to <em>compile</em> for. This specifies what <em>capabilities</em> the code can use (features, etc). Default value: <code>compute_10</code></td>
</tr>
<tr>
<td><code>--gpu-code &lt;gpu&gt;</code></td>
<td>-code</td>
<td>Specify the GPU(s) to <em>generate code</em> for, i.e. what PTX assembly and/or binary code to generate. Default value: <code>compute_10,sm_10</code></td>
</tr>
<tr>
<td><code>--generate-code</code></td>
<td>-gencode</td>
<td>Generalize -arch and -code into a single option with keywords for convenience. -gencode arch=… code=…</td>
</tr>
</tbody>
</table>
JIT Caching

- To mitigate JIT startup penalty, the NVIDIA driver does cache the most recent binary code generated by JIT.
- However, this is stored by default on disk and in the user’s homedir, so this can lead to additional latency on slow network shares.
- JIT Caching behavior is configurable:

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_CACHE_DISABLE</td>
<td>Disables JIT Caching if set to 1</td>
</tr>
<tr>
<td>CUDA_CACHE_MAXSIZE</td>
<td>Maximum size of the user’s JIT cache. Default is 32 MB, max is 4 GB.</td>
</tr>
<tr>
<td>CUDA_CACHE_PATH</td>
<td>Storage location of the JIT cache. Defaults to ~/.nv/ComputeCache on Linux</td>
</tr>
<tr>
<td>CUDA_FORCE_PTX_JIT</td>
<td>Force the driver to ignore any binary code in the application and JIT from PTX. If there is no PTX, this will fail.</td>
</tr>
</tbody>
</table>
Linking to libcudart.so

- CUDA runtime library may have breaking changes between versions, so nvcc links against a particular version of libcudart.so:

  $ nvcc cuda-hello.cu
  $ ldd a.out
  libcudart.so.5.0 => /opt/cuda-5.0/lib64/libcudart.so.5.0

  ...

- This means that if an application is compiled with a particular runtime, that library will need to be available for it to run

- CUDA 5.5 now supports compiling the CUDA runtime statically in the application, and static compilation is now the default

- New nvcc option to configure this behavior:

  nvcc --cudart={static,shared,none}
Host compiler compatibility

- Host compiler in NVCC is g++ (uses first one in PATH)

- If you want to use a different compiler with CUDA (Intel, PGI, etc), need to be able to link against GCC ABI

- Best practice:
  - Minimize performance-critical host code in files processed by nvcc
  - Link with objects produced by your compiler of choice

- Common pattern: build shared library containing all CUDA code, link to it from your larger application
Other ways to build CUDA apps
GPU Accelerated Libraries
“Drop-in” Acceleration for your Applications

- NVIDIA cuBLAS
- NVIDIA cuRAND
- NVIDIA cuSPARSE
- NVIDIA NPP
- GPU VSIPL
- GPU Accelerated Linear Algebra
- MAGMA
- Matrix Algebra on GPU and Multicore
- NVIDIA cuFFT
- Rogue Wave
- Sparse Linear Algebra
- CUSP
- Building-block Algorithms
- Thrust
- C++ Templated Parallel Algorithms
GPU Accelerated Libraries
“Drop-in” Acceleration for your Applications

NVIDIA cuBLAS
NVIDIA cuRAND
NVIDIA cuSPARSE
NVIDIA NPP

GPU VSIPL
cuAccelerometer tools
MAGMA
NVIDIA cuFFT

Included in CUDA Toolkit

IMSL Library
Sparse Linear Algebra
Building-block Algorithms
C++ Templated Parallel Algorithms

Copyright © NVIDIA Corporation
OpenACC Directives

Your original Fortran or C code

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

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs
OpenACC

- Useful way to quickly add CUDA support to a program without writing CUDA code directly, especially for legacy apps
- Uses compiler directives very similar to OpenMP
- Supports C and Fortran
- Generally doesn’t produce code as fast as a good CUDA programmer… but often get decent speedups
- Cross-platform; depending on compiler, supports NVIDIA, AMD, Intel accelerators

Compiler support:
- Cray 8.0+
- PGI 12.6+
- CAPS HMPP 3.2.1+

http://developer.nvidia.com/openacc
$ pgcc -acc -Minfo=accel -ta=nvidia -o saxpy_acc saxpy.c
PGC-W-0095-Type cast required for this conversion (saxpy.c: 13)
PGC-W-0155-Pointer value created from a nonlong integral type (saxpy.c: 13)
saxpy:
  4, Generating present_or_copyin(x[0:n])
  Generating present_or_copy(y[0:n])
  Generating NVIDIA code
  Generating compute capability 1.0 binary
  Generating compute capability 2.0 binary
  Generating compute capability 3.0 binary
  5, Loop is parallelizable
  Accelerator kernel generated
    5, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
PGC/x86-64 Linux 13.2-0: compilation completed with warnings
OpenACC

- PGI compiler generates...
  - Object code for currently-installed GPU, if supported (auto-detect)
  - PTX assembly for all major versions (1.0, 2.0, 3.0)

- Depending on the compiler step, there may or may not be a OpenACC->CUDA C translation step before compile (but this intermediate code is usually not accessible)
CUDA Fortran

- Slightly-modified Fortran language which uses the CUDA Runtime API
- Almost 1:1 translation of CUDA C concepts to Fortran 90
- Changes mostly to conform to Fortran idioms (“Fortranic”?)

- Currently supported only by PGI Fortran compiler
- pgfortran acts like “nvcc for Fortran” with either the -Mcuda option, or if you use the file extension .cuf
- Compiles to CUDA C as intermediate. Can keep C code with option “-Mcuda=keepgpu”
Other GPU Programming Languages

**Numerical analytics**
- MATLAB, Mathematica, LabVIEW

**Fortran**
- OpenACC, CUDA Fortran

**C**
- OpenACC, CUDA C

**C++**
- Thrust, CUDA C++

**Python**
- PyCUDA, Copperhead, Numba Pro

**C#**
- GPU.NET
Other GPU Programming Languages

- Current version of NVCC uses LLVM internally

- Code to compile LLVM IR to PTX assembly is open source (BSD license), so adding additional language support is easier


- Compiler SDK is also included by default in 5.5 Toolkit: $CUDA_HOME/nvvm
Other Resources

CUDA Toolkit Documentation: http://docs.nvidia.com

OpenACC: http://www.openacc.org/

CUDA Fortran @ PGI: http://www.pgroup.com/resources/cudafortran.htm

GPU Applications Catalog (list of known common apps with GPU support): http://www.nvidia.com/docs/IO/123576/nv-applications-catalog-lowres.pdf

Email me! Adam DeConinck, adeconinck@nvidia.com

...and many other resources available via CUDA Registered Developer program. https://developer.nvidia.com/nvidia-registered-developer-program
Upcoming GTC Express Webinars

**July 11** - Uncovering the Elusive HIV Capsid with Kepler GPUs Running NAMD and VMD

**July 17** - Delivering 3D Graphics from the Private or Public Cloud with XenDesktop and GRID

**July 25** - ACEMD: High-throughput Molecular Dynamics with NVIDIA Kepler GPUs

**July 30** - Getting Started with GPU-accelerated Computer Vision using OpenCV and CUDA

**July 31** - NMath Premium: GPU-Accelerated Math Libraries for .NET

Register at www.gputechconf.com/gtcexpress
GTC 2014 Call for Submissions

Looking for submissions in the fields of

- Science and research
- Professional graphics
- Mobile computing
- Automotive applications
- Game development
- Cloud computing

Submit at www.gputechconf.com