Introduction to the CUDA Toolkit for Building Applications

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***

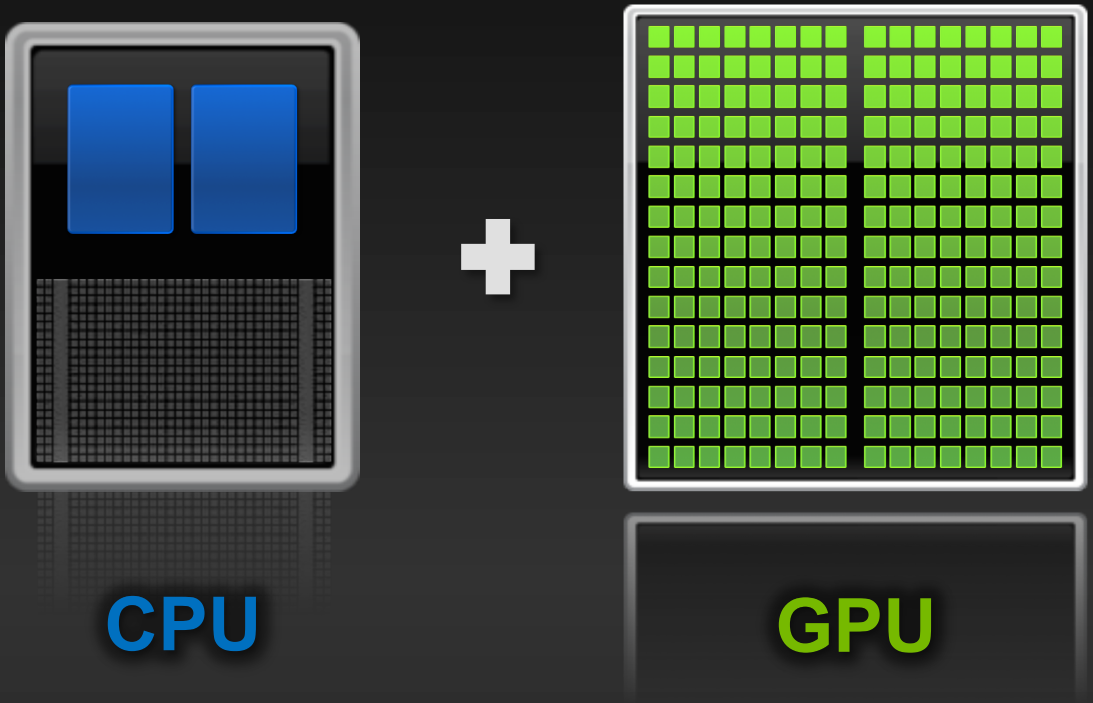
  - Review GPU Computing concepts
  - CUDA C/C++ with nvcc compiler
  - Example application build processes
  - OpenACC compilers
  - Common libraries

- **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

Copyright © NVIDIA Corporation

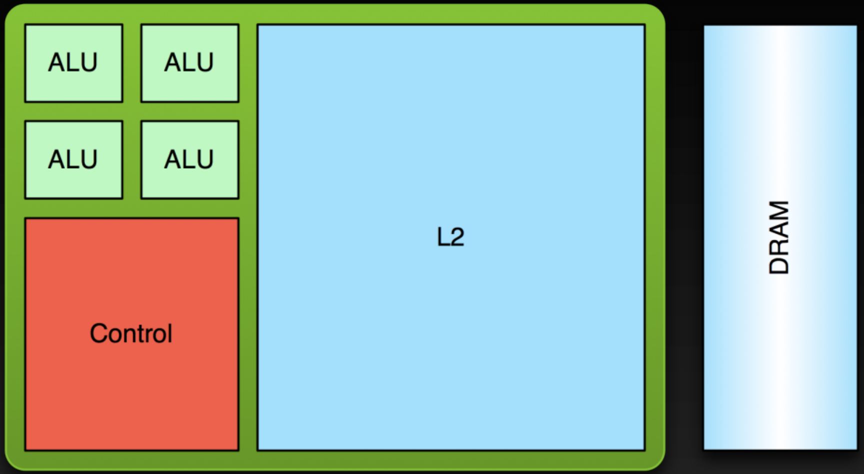
CPU vs GPU

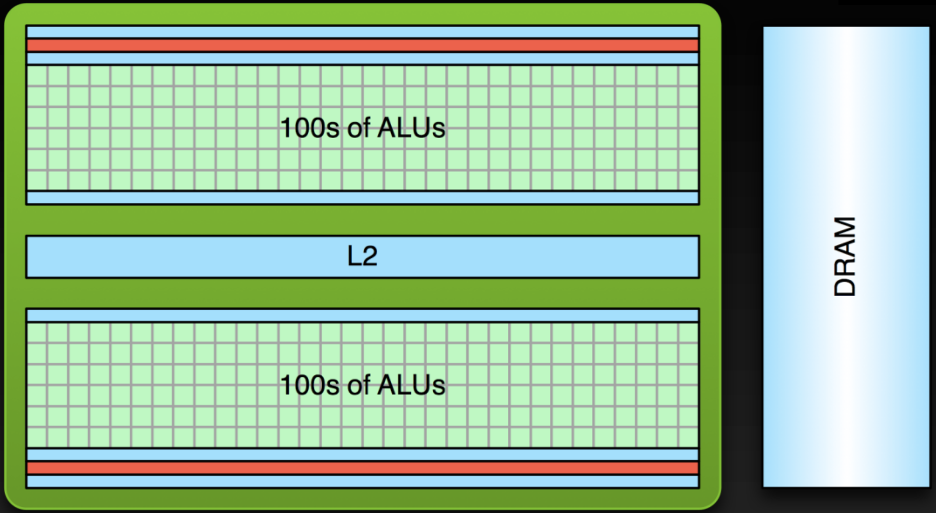
Latency Processor + Throughput processor



Copyright © NVIDIA Corporation

Low Latency or High Throughput?





CPU

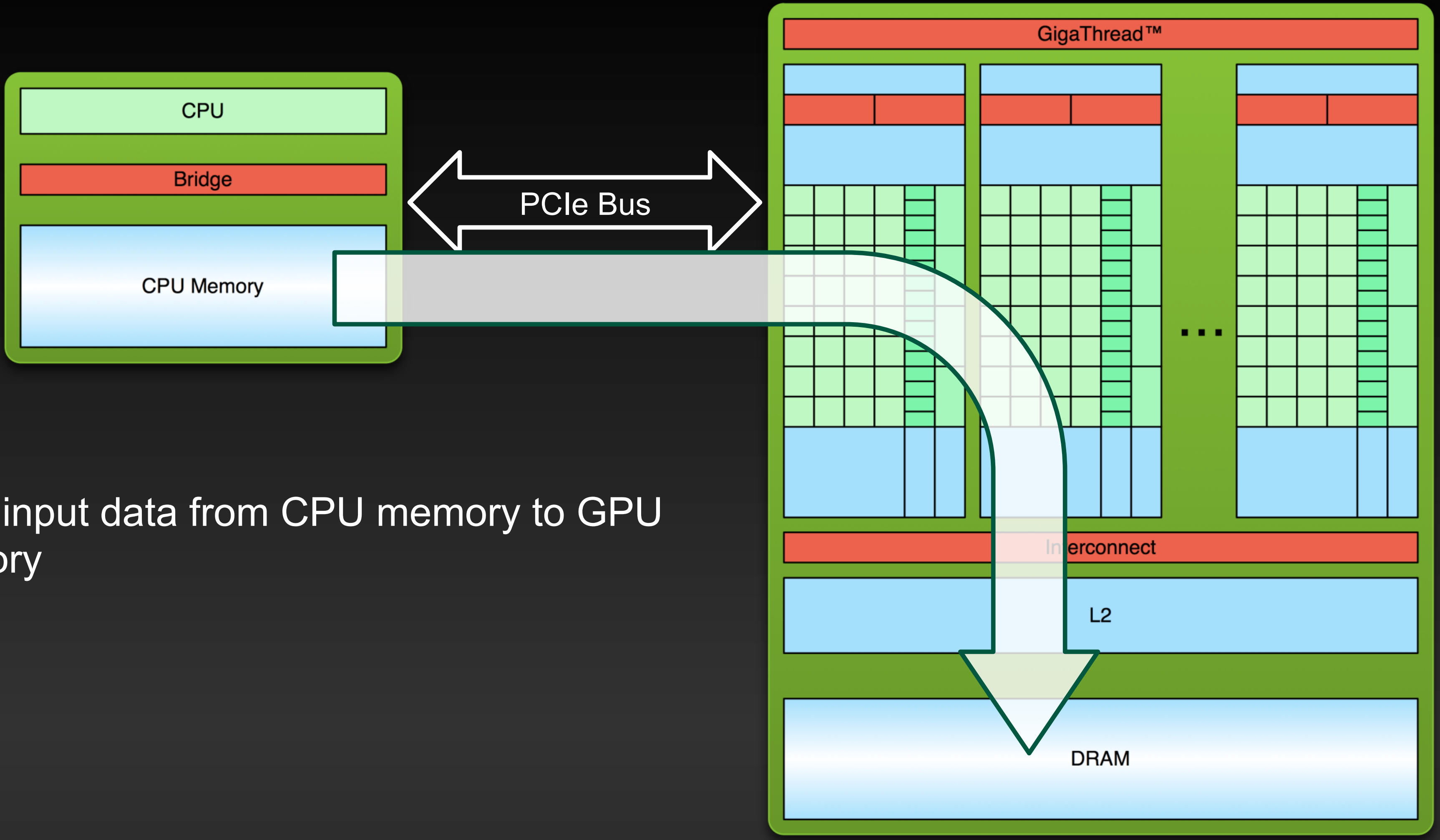
- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

GPU

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

Copyright © NVIDIA Corporation

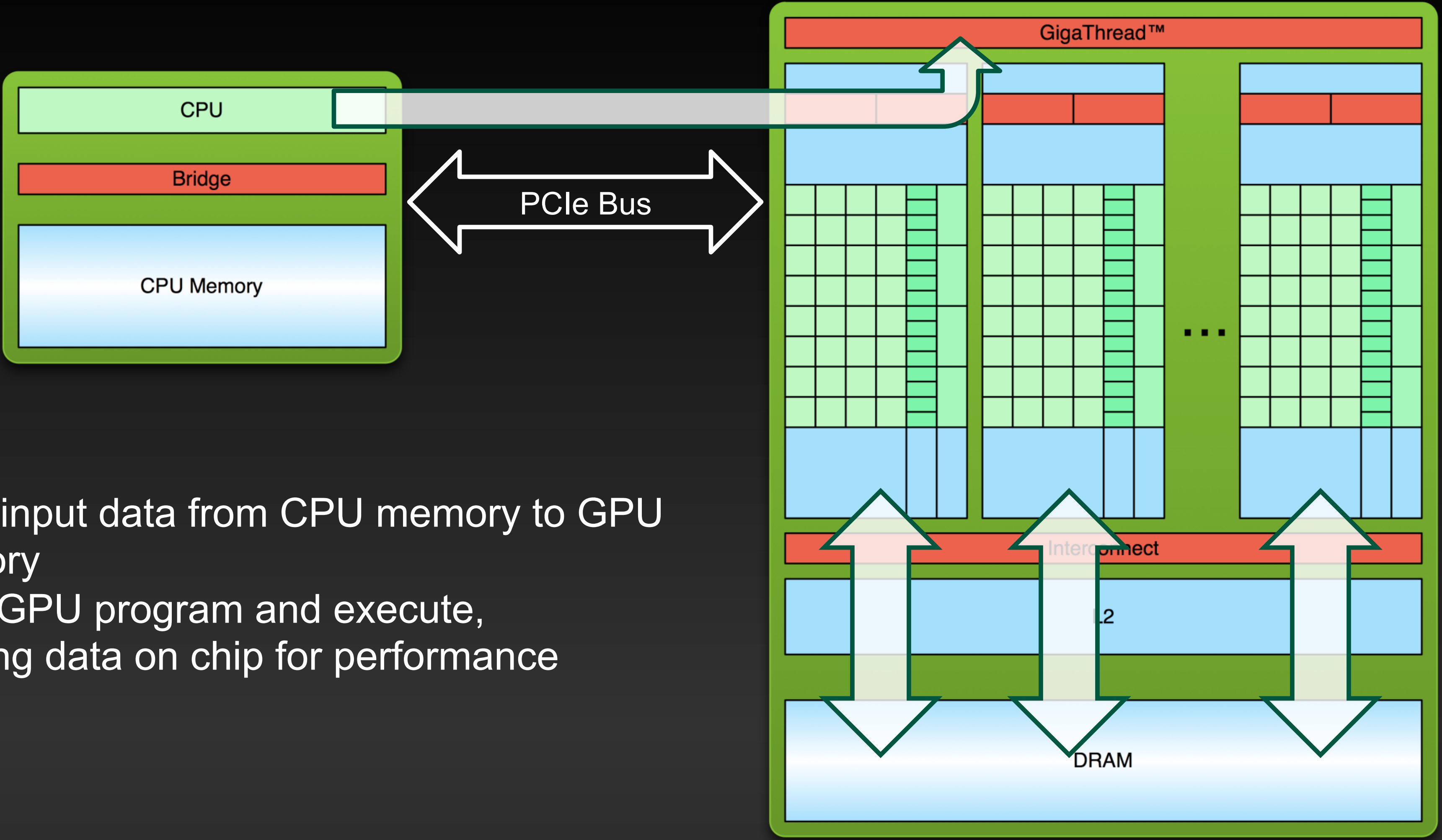
Processing Flow



1. Copy input data from CPU memory to GPU memory

Copyright © NVIDIA Corporation

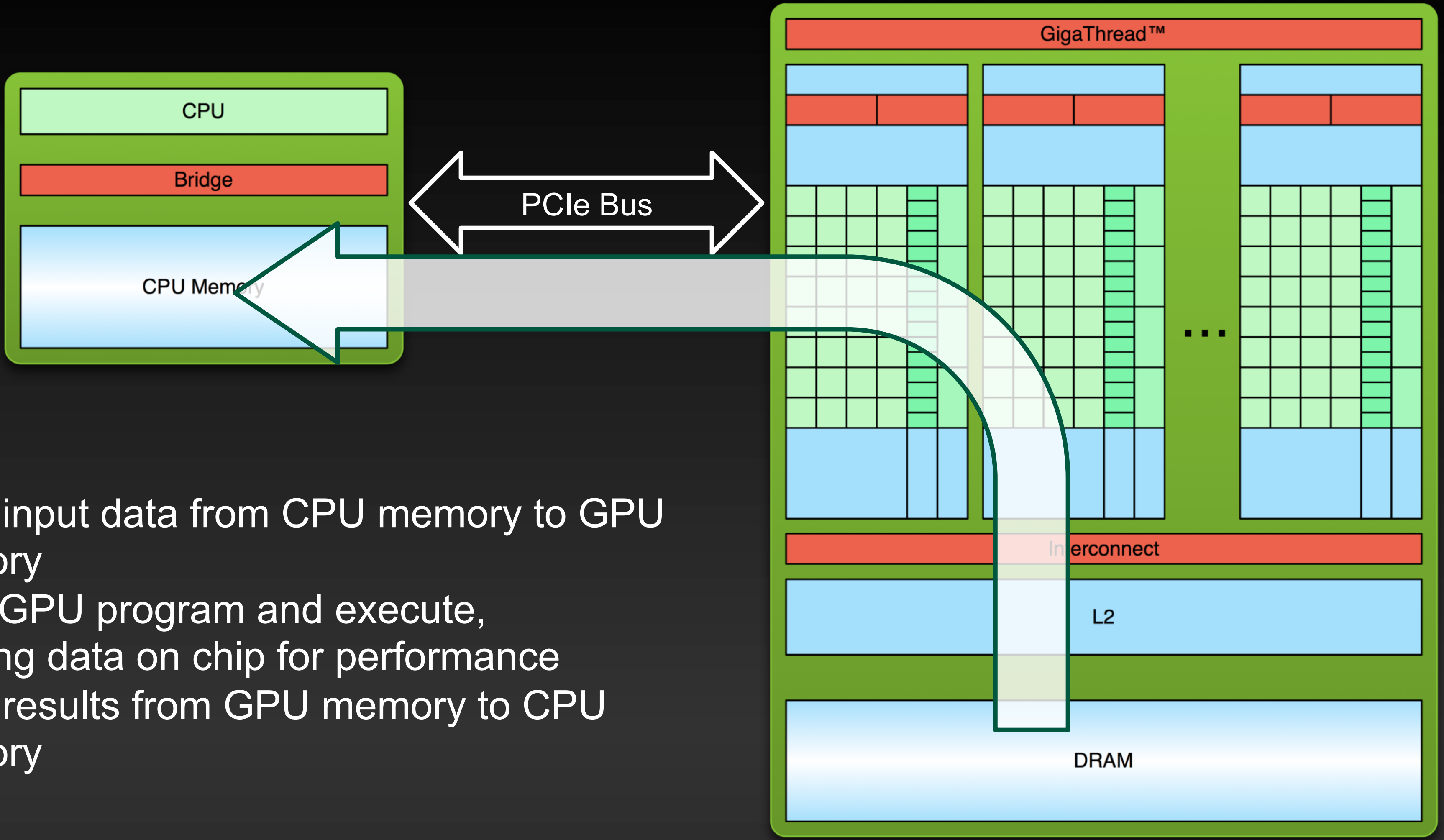
Processing Flow



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

Copyright © NVIDIA Corporation

Processing Flow

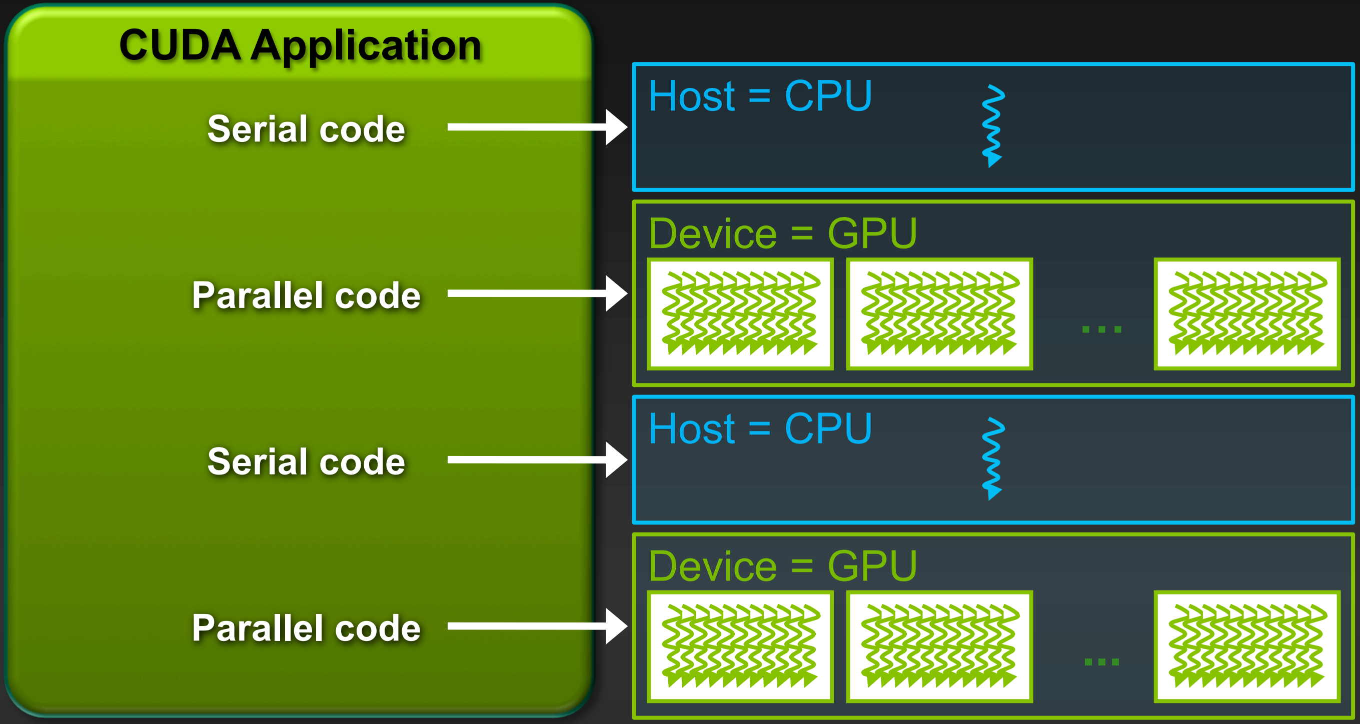


1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Copyright © NVIDIA Corporation

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



Copyright © NVIDIA Corporation

CUDA C

Standard 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];
}

// Perform SAXPY on 1M elements
saxpy_serial(4096*256, 2.0, x, y);
```

Parallel C Code

```
__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];
}

// Perform SAXPY on 1M elements
saxpy_parallel<<<4096,256>>>(n,2.0,x,y);
```

Copyright © NVIDIA Corporation

3 Ways to Accelerate Applications

Applications

Libraries	OpenACC Directives	Programming Languages
"Drop-in" Acceleration	Compiler directives (Like OpenMP)	Most common: CUDA C Also CUDA Fortran, PyCUDA, Matlab, …

Copyright © NVIDIA Corporation

3 Ways to Accelerate Applications

Applications

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

Programming Languages

Most common: CUDA C

Also CUDA Fortran, PyCUDA, Matlab, …

Copyright © NVIDIA Corporation

The CUDA Toolkit

Copyright © NVIDIA Corporation

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

Copyright © NVIDIA Corporation

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
```

Copyright © NVIDIA Corporation

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 to MPI in that you just have to point to nvcc correctly (like pointing to the right mpicc)
  - But you always have to use the "special" 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

Copyright © NVIDIA Corporation

Where’s CUDA?

Common to install CUDA somewhere other than /usr/local/cuda, so where is it?

- 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.

Copyright © NVIDIA Corporation

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

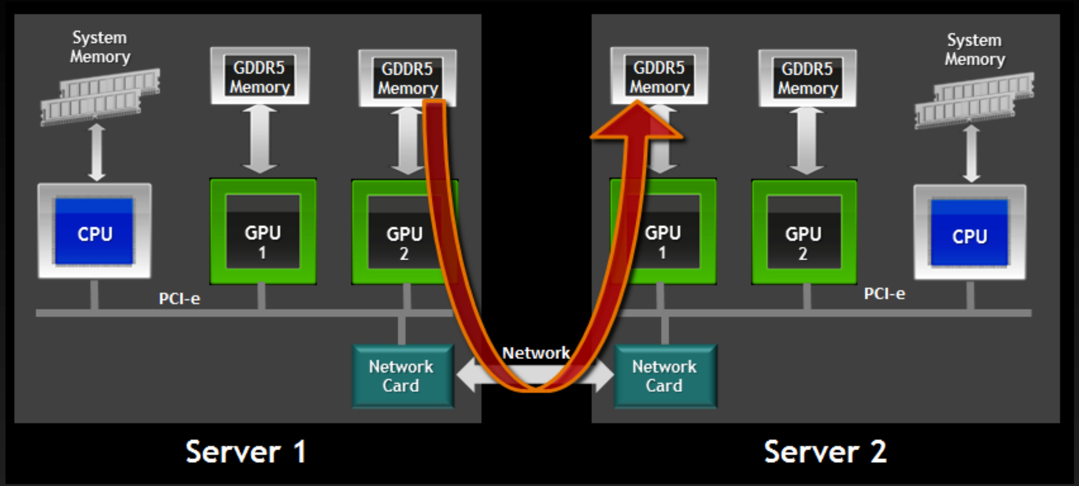
Copyright © NVIDIA Corporation

Common NVCC Options

Environment variable	Command-line flag	Equivalent for gcc	Definition
INCLUDES	--include-path -I	CPATH -I	Define additional include paths
LIBRARIES	--library-path -L	LD_LIBRARY_PATH -L	Define additional library paths
	--optimize -O	-O	Optimization level for host code
	-use_fast_math		Apply all device-level math optimizations
PTXAS_FLAGS	-Xptxas=-v		Print GPU resources (shared memory, registers) used per kernel

Copyright © NVIDIA Corporation

CUDA support in MPI implementations



- 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

Copyright © NVIDIA Corporation

Example Builds

Copyright © NVIDIA Corporation

Example: matrixMul

- Part of the CUDA 5 Samples (distributed with CUDA Toolkit)
- 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
...
```

Copyright © NVIDIA Corporation

Example: simpleMPI

- Part of the CUDA 5 Samples (distributed with CUDA Toolkit)
- 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

Copyright © NVIDIA Corporation

```
$ 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
```

Copyright © NVIDIA Corporation

```
$ 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
```

(we'll explain the -gencode bits later)

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

Copyright © NVIDIA Corporation

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)
```

Copyright © NVIDIA Corporation

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 >= 2.0 (Fermi or Kepler)

Copyright © NVIDIA Corporation

Example: GROMACS

```
wget ftp://ftp.gromacs.org/pub/gromacs/gromacs-4.6.tar.gz
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 -
DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=./gromacs-build

make install
```

Copyright © NVIDIA Corporation

Example: GROMACS (cmake)

- cmake defines a number of environment variables for controlling nvcc compiler

Environment variables	Meaning
CUDA_HOST_COMPILER	Specify which host-code compiler to use (i.e. which gcc)
CUDA_HOST_COMPILER_OPTIONS	Options passed to the host compiler
CUDA_NVCC_FLAGS	Options passed to nvcc

- 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;
```

Copyright © NVIDIA Corporation

NVCC Build Process

Copyright © NVIDIA Corporation

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

Copyright © NVIDIA Corporation

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 slight startup lag)
  - If neither version are compatible with the GPU, the application doesn’t run.
- Always uses the correct object code, or the newest compatible PTX

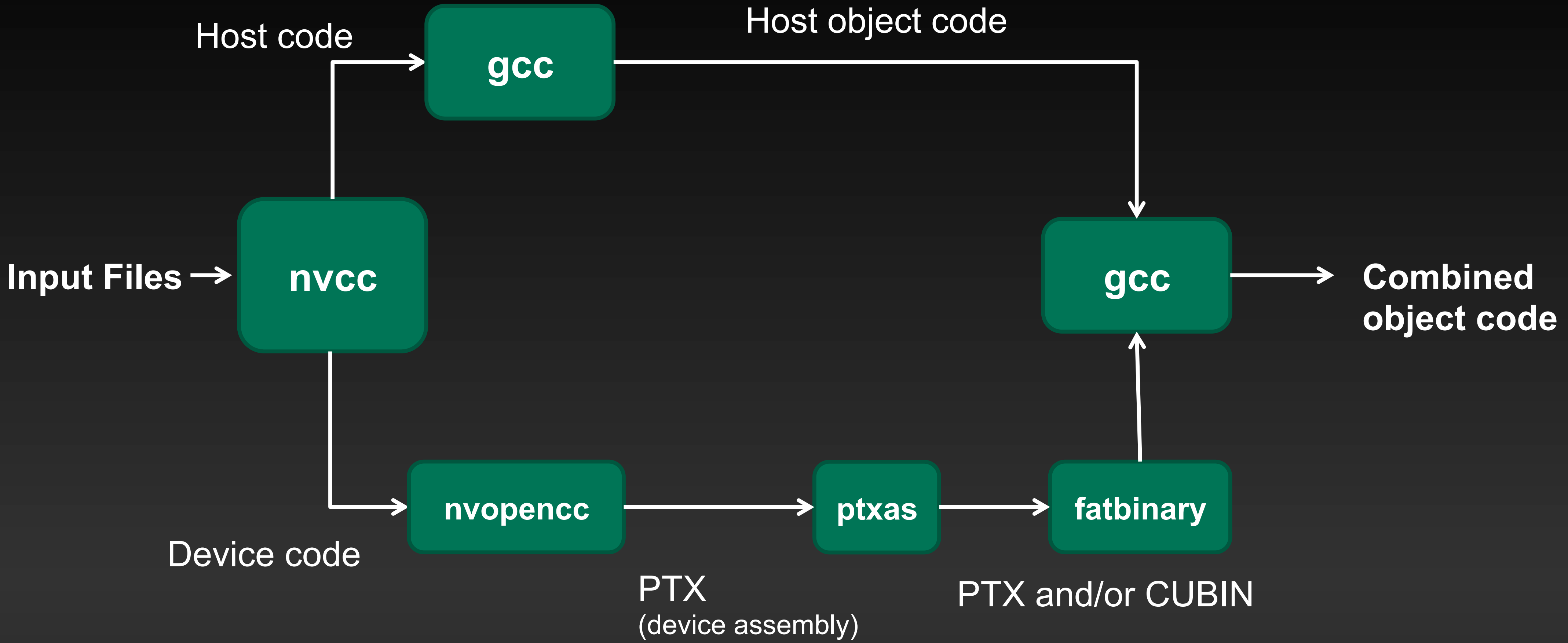
Copyright © NVIDIA Corporation

Why do we care?

- A given CUDA binary is not guaranteed to run on an arbitrary GPU
- And if it does run, not guaranteed to get best performance
  - JIT startup time
  - 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

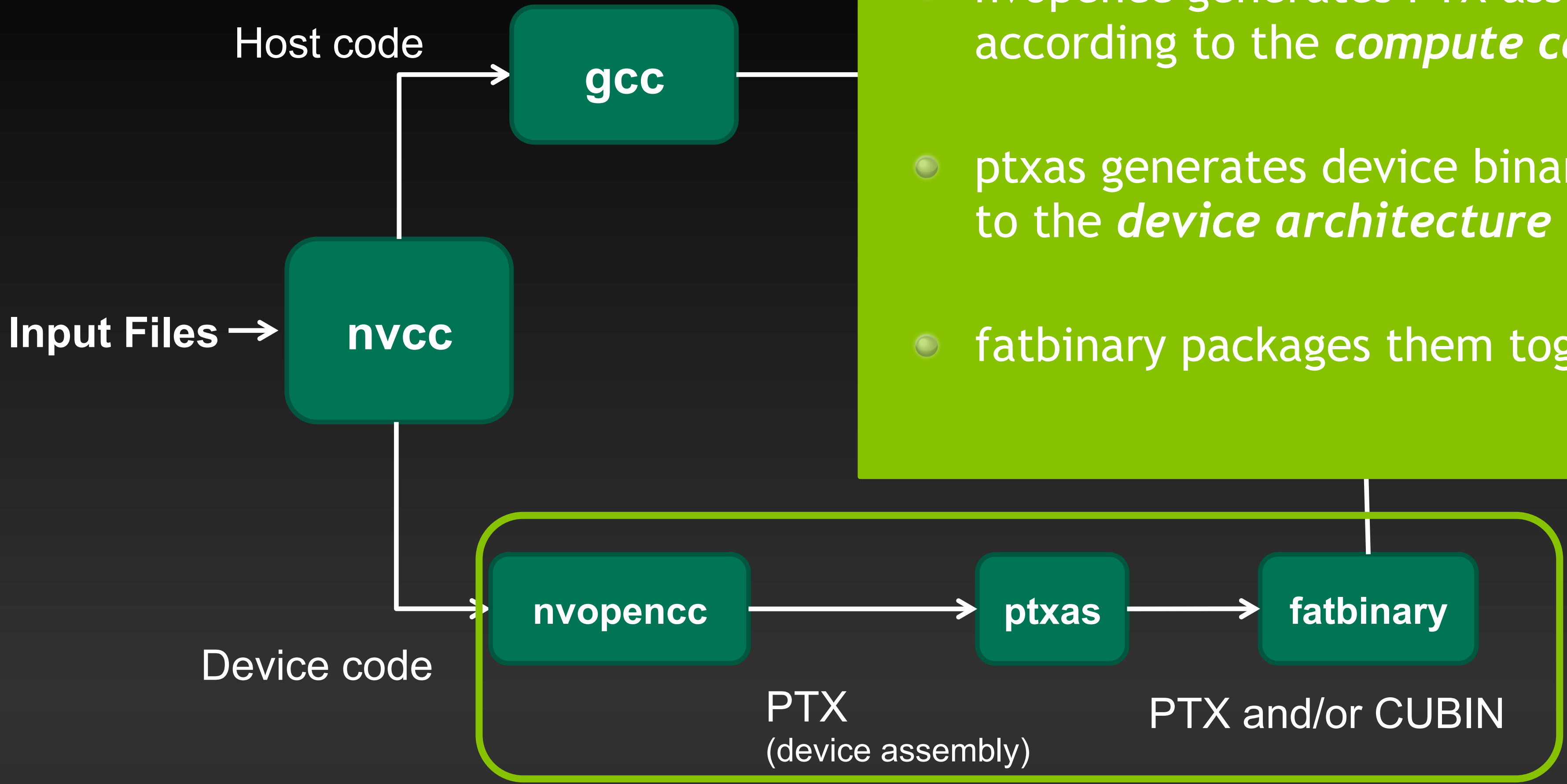
Copyright © NVIDIA Corporation

NVCC Build Process (simplified)



Copyright © NVIDIA Corporation

NVCC Build Process (simplified)



- nvopencc generates PTX assembly according to the ***compute capability***
- ptxas generates device binaries according to the ***device architecture***
- fatbinary packages them together

Copyright © NVIDIA Corporation

Options to different stages

Environment variables	Command-line options	Meaning
	-Xcompiler	Pass options directly to the (host) compiler/preprocessor (i.e. gcc)
	-Xlinker	Pass options directly to the linker
	-Xcudafe	Pass options directly to cudafe (pre-processor/splitter)
OPENCC_FLAGS	-Xopencc	Pass options directly to nvopencc, typically for steering device code optimization
PTXAS_FLAGS	-Xptxas	Pass options directly to the ptx optimizing compiler

Copyright © NVIDIA Corporation

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)

  - PTX Assembly version
  - 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

Copyright © NVIDIA Corporation

Compute capability and device architecture

- When you compile code with NVCC, you can specify
  - Compute capabilities, which describe version of CUDA language & PTX. I.e., *compute_20*.
  - Device architectures, which describe version of CUDA object code. I.e., *sm_20*.
- 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
```

Copyright © NVIDIA Corporation

Command line options for specifying arch

Long option	Short option	Meaning
--gpu-architecture <*arch*>	-arch	Specify the GPU architecture to *compile* for. This specifies what *capabilities* the code can use (features, etc). Default value: compute_10
--gpu-code <*gpu*>	-code	Specify the GPU(s) to *generate code* for, i.e. what PTX assembly and/or binary code to generate. Default value: compute_10,sm_10
--generate-code	-gencode	Generalize -arch and -code into a single option with keywords for convenience. -gencode arch=… code=…

Copyright © NVIDIA Corporation

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;-
  gencode;arch=compute_30,code=compute_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

Copyright © NVIDIA Corporation

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=compute_13,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=compute_30,sm_30`

Copyright © NVIDIA Corporation

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

Copyright © NVIDIA Corporation

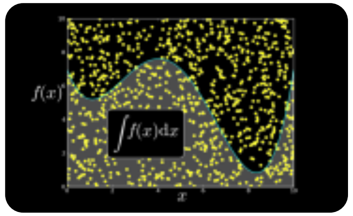
Libraries and Other Compilers

Copyright © NVIDIA Corporation

GPU Accelerated Libraries

“Drop-in” Acceleration for your Applications

NVIDIA cuBLAS



NVIDIA cuRAND

NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

Vector Signal Image Processing

CULA | tools

GPU Accelerated Linear Algebra

MAGMA

Matrix Algebra on GPU and Multicore

NVIDIA cuFFT

ROGUE WAVE SOFTWARE

IMSL Library

CUSP

Sparse Linear Algebra

ArrayFire

Building-block Algorithms

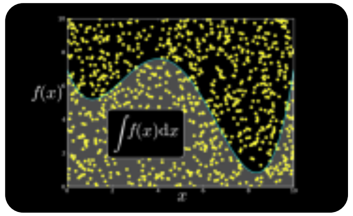
Thrust

C++ Templated Parallel Algorithms

Copyright © NVIDIA Corporation

GPU Accelerated Libraries
“Drop-in” Acceleration for your Applications

NVIDIA cuBLAS



NVIDIA cuRAND

NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

CULA tools

ed

a

MAGMA

ICL UT

Matrix Algebra on GPU and Multicore

NVIDIA cuFFT

Included in CUDA Toolkit

ROGUE WAVE SOFTWARE

IMSL Library

CUSP

Sparse Linear Algebra

ArrayFire

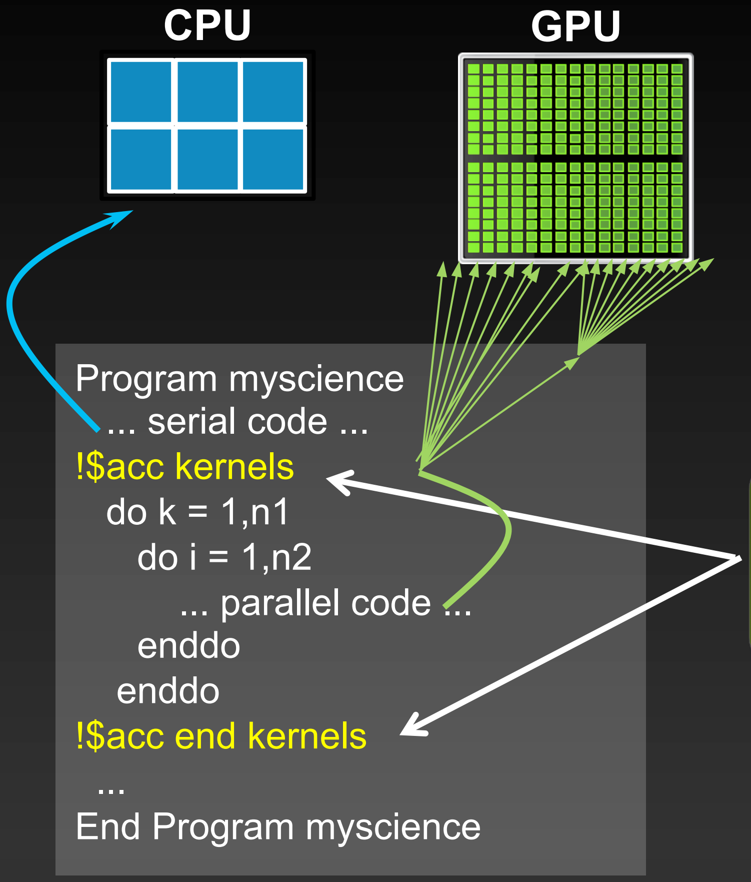
Building-block Algorithms

Thrust

C++ Templated Parallel Algorithms

Copyright © NVIDIA Corporation

OpenACC Directives



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

Your original
Fortran or C code

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

Copyright © NVIDIA Corporation

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

Copyright © NVIDIA Corporation

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
```

Copyright © NVIDIA Corporation

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)

Copyright © NVIDIA Corporation

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”

Copyright © NVIDIA Corporation

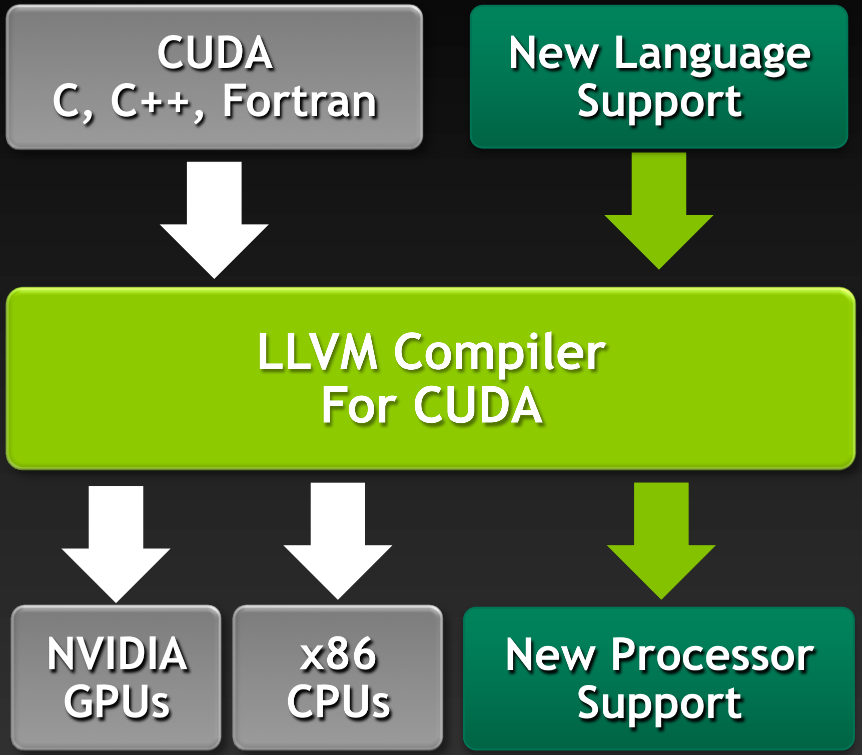
Other GPU Programming Languages

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

Copyright © NVIDIA Corporation

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
- More information: Compiler SDK https://developer.nvidia.com/cuda-llvm-compiler



Copyright © NVIDIA Corporation

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

Copyright © NVIDIA Corporation

Questions?

Copyright © NVIDIA Corporation

ISV Applications

- …or maybe you don't have to build the application at all! If using an ISV application, distributed as a binary.

- Important to be careful about libraries for pre-compiled packages, especially CUDA Runtime:
  - Many applications distribute a particular libcudart.so
  - Dependent on that particular version, may break with later versions
  - Apps don't always link to it intelligently; be careful with your modules!

Copyright © NVIDIA Corporation

Driver API vs Runtime API

- CUDA GPUs expose two APIs: “driver API” and “runtime API”

- Driver API is much more complex, but provides more control over low-level details. Link directly to the driver’s libcuda.so.
- Driver API applications are not necessarily forward compatible

- Runtime API is much simpler, and is the “CUDA language” most people think of.
- Compiled with NVCC, programs link runtime library (libcudart.so)

- Vastly more programs use runtime API, so we’ll focus on that

Copyright © NVIDIA Corporation