Introduction to GPGPU Programming

Pragnesh Patel
pragnesh@utk.edu
Remote Data Analysis and Visualization Center
National Institute for Computational Sciences
University of Tennessee

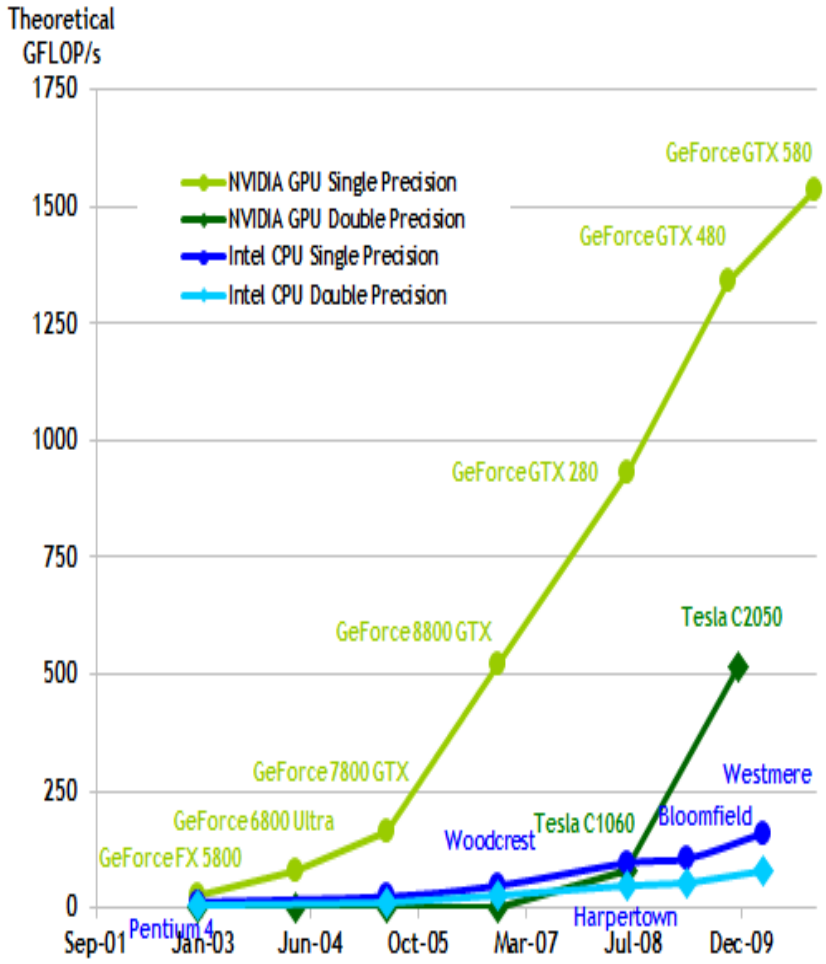
NICS

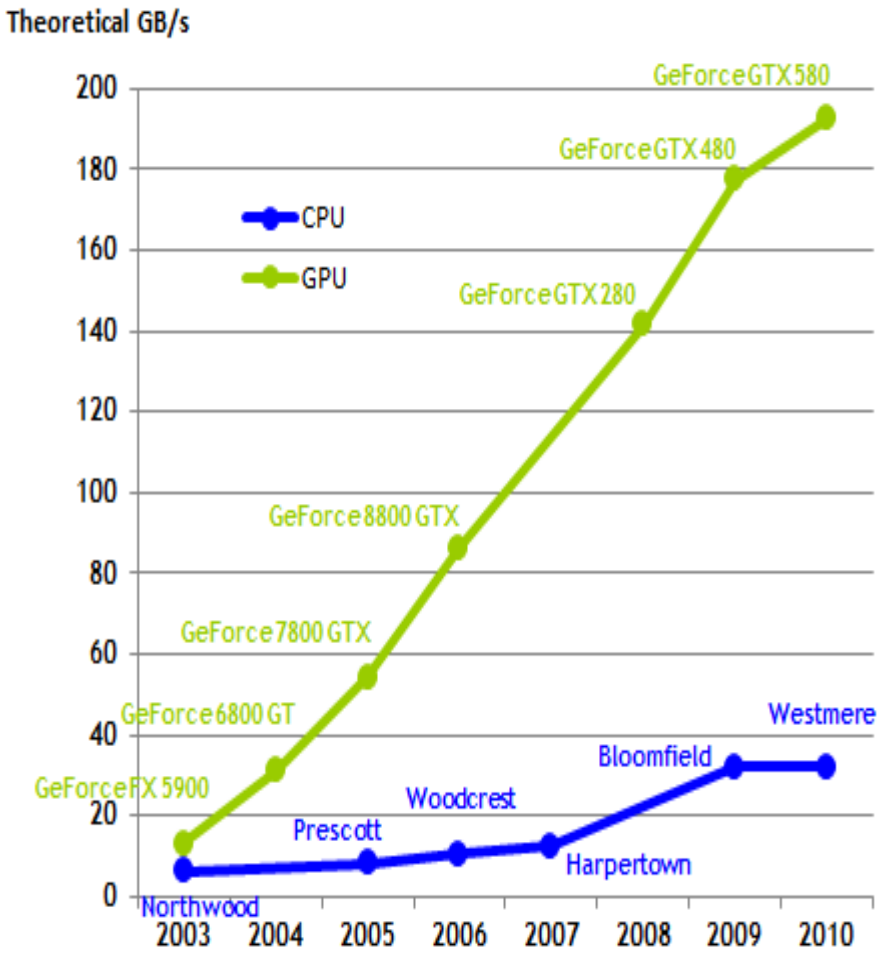
RDAV

Outline:

- Motivation
- History
- GPU architecture
- GPU programming model
- CUDA C
- CUDA tools
- OpenCL in brief
- Other useful GPU tools
- Summary
- References

Motivation:


Theoretical GFLOP/s
1750
1500
1250
1000
750
500
250
0
NVIDIA GPU Single Precision
NVIDIA GPU Double Precision
Intel CPU Single Precision
Intel CPU Double Precision
GeForce GTX 580
GeForce GTX 480
GeForce GTX 280
GeForce 8800 GTX
GeForce 7800 GTX
GeForce 6800 Ultra
GeForce FX 5800
Tesla C2050
Tesla C1060
Westmere
Bloomfield
Woodcrest
Harpertown
Pentium 4
Sep-01
Jan-03
Jun-04
Oct-05
Mar-07
Jul-08
Dec-09


Theoretical GB/s
200
180
160
140
120
100
80
60
40
20
0
CPU
GPU
GeForce GTX 580
GeForce GTX 480
GeForce GTX 280
GeForce 8800 GTX
GeForce 7800 GTX
GeForce 6800 GT
GeForce FX 5900
Westmere
Bloomfield
Woodcrest
Prescott
Harpertown
Northwood
2003
2004
2005
2006
2007
2008
2009
2010

History:

- Graphics Processing Unit
  - Designed to rapidly manipulate and alter memory in such a way so as to accelerate the building of images in a frame buffer intended for output to a display.
  - The term was popularized by NVIDIA in 1999.
  - GeForce 256: The World's first GPU.

GPU:

- GPU = Graphics Processing Unit
  - Chip in computer video cards, PlayStation 3, XBOX etc..
  - Two major vendors: NVIDIA and AMD

Reference: http://en.wikipedia.org/wiki/Nvidia_Tesla

NVIDIA GPUs:

- Desktop GPUs
  - GeForce series for CPU
- Mobile GPUs
  - GeForce series for Mobile
- Workstation/HPC GPUs
  - Quadro NVS, Tesla, Kepler

NVIDIA GPUs:

- Supports CUDA and OpenCL
- Fermi(Tesla version)
  - Upto 512 cores
  - Upto 6GB memory
  - Upto 665 GFLOPS – Double precision
  - Caches included: L1 per multiprocessor, L2 shared
- Kepler in 2012
- Maxwell in 2014

AMD GPUs:

- Desktop GPUs
  - Radeon series
- Mobile GPUs
  - Mobility Radeon
- Workstation GPUs
  - FirePro, FireStream
- Supports OpenCL (no CUDA)

GPU continue:

- Modern GPUs are very efficient at
  - Manipulating computer graphics, and their highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing of large block of data is done in parallel.
- GPUs are massively multithreaded manycore chips.
  - NVIDIA tesla products have upto 512 cores.
  - Over 665 GFLOPS sustained performance (double floating point)
  - 6GB of Memory
  - Memory bandwidth upto 177 GBytes/sec.

- Users across science and engineering disciplines are achieving very good speedups on GPUs.

Typical Supercomputer:

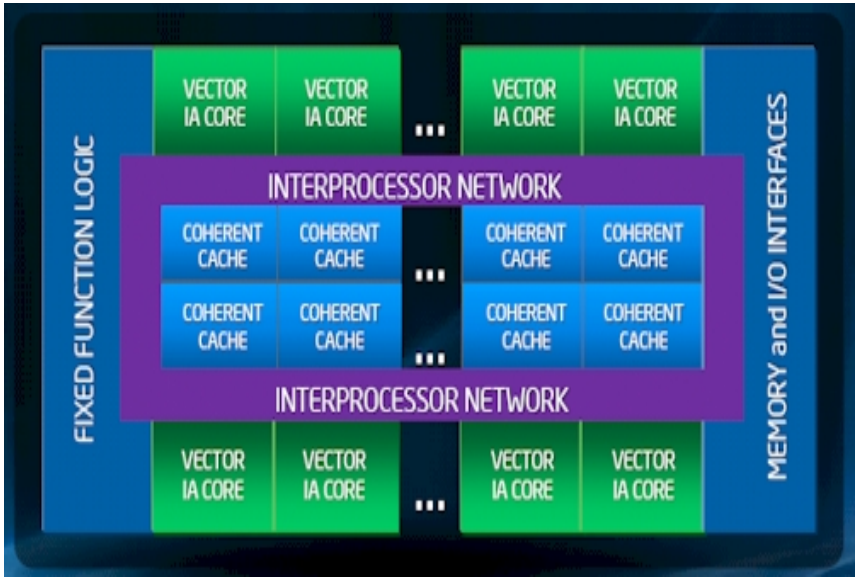
- Large amount of nodes.
  - Distributed memory
  - Multicore processors (e.g. 12 cores per node Kraken)
- Fast interconnect.
- Programming models
  - MPI
  - Hybrid (Pthreads, OpenMP with MPI)

Accelerated Supercomputer:

- Accelerated HPC floating workloads using GPUs.
  - Peak FP performance 10x vs CPU.
  - Memory bandwidth 20x vs CPU.
  - Parallelism, of the order of 500 cores, thousands of threads.
- GPUs are accelerators.
  - Has its own fast memory.
  - Separate card connected to CPU node Via PCI-E bus.

Other accelerators:

- Intel
  - Intel MIC(Many Integrated Core)
  - ~ 50 X86 vector cores
  - OpenMP, OpenCL, Intel parallel building blocks etc...
  - First commercial product(Knights corner) in 2012.
- Other(FPGA and DSP based system etc…)



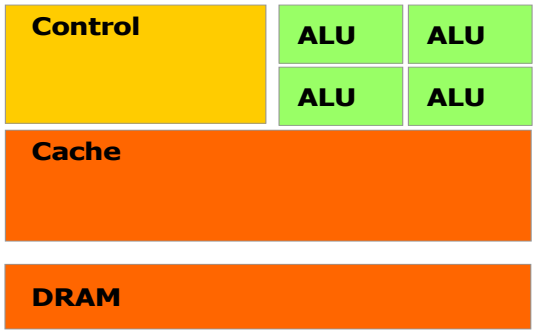
Reference: http://intel.com

Not for all applications:

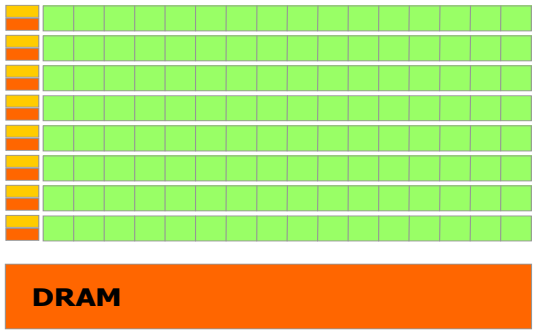
- SIMD(Single Program, Multiple Data) are best.
- Operations need to be sufficient size to overcome overhead.
- Think millions of operations.
- Data transfer could be bottleneck (between CPU memory and GPU memory)

How it is different from CPU:

- GPU is specialized for compute-intensive, highly parallel computation-exactly what graphics rendering is about.
- GPU devotes more transistors to data processing rather than data caching and flow control.

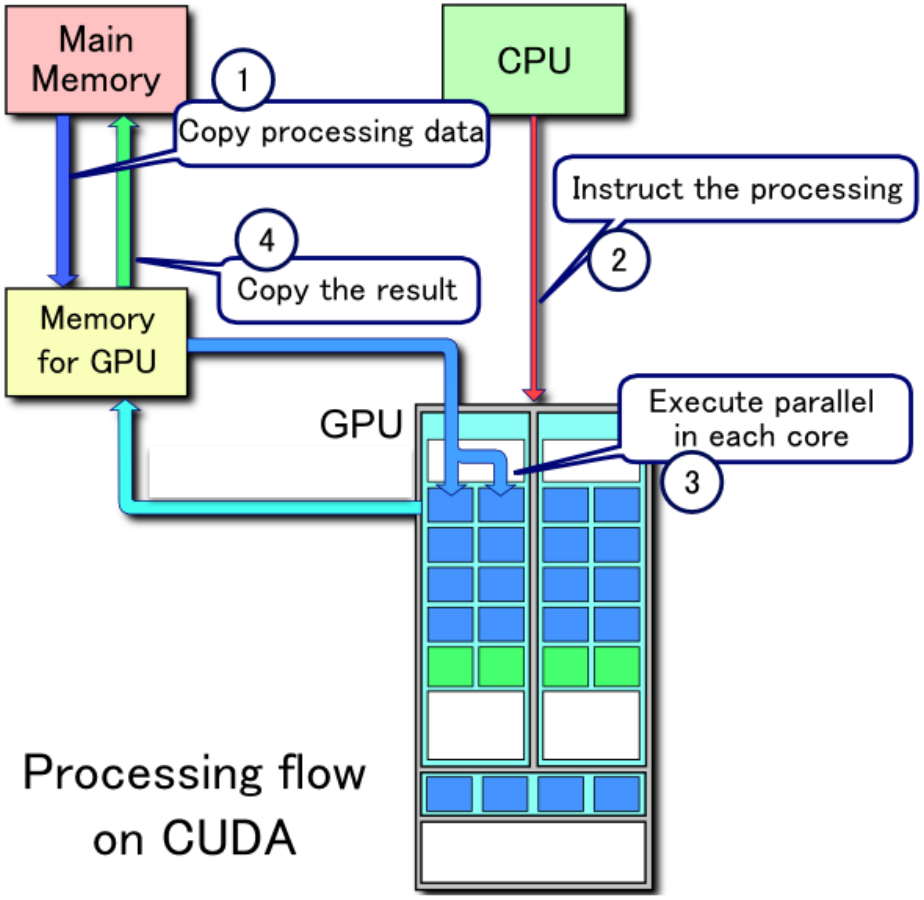


CPU



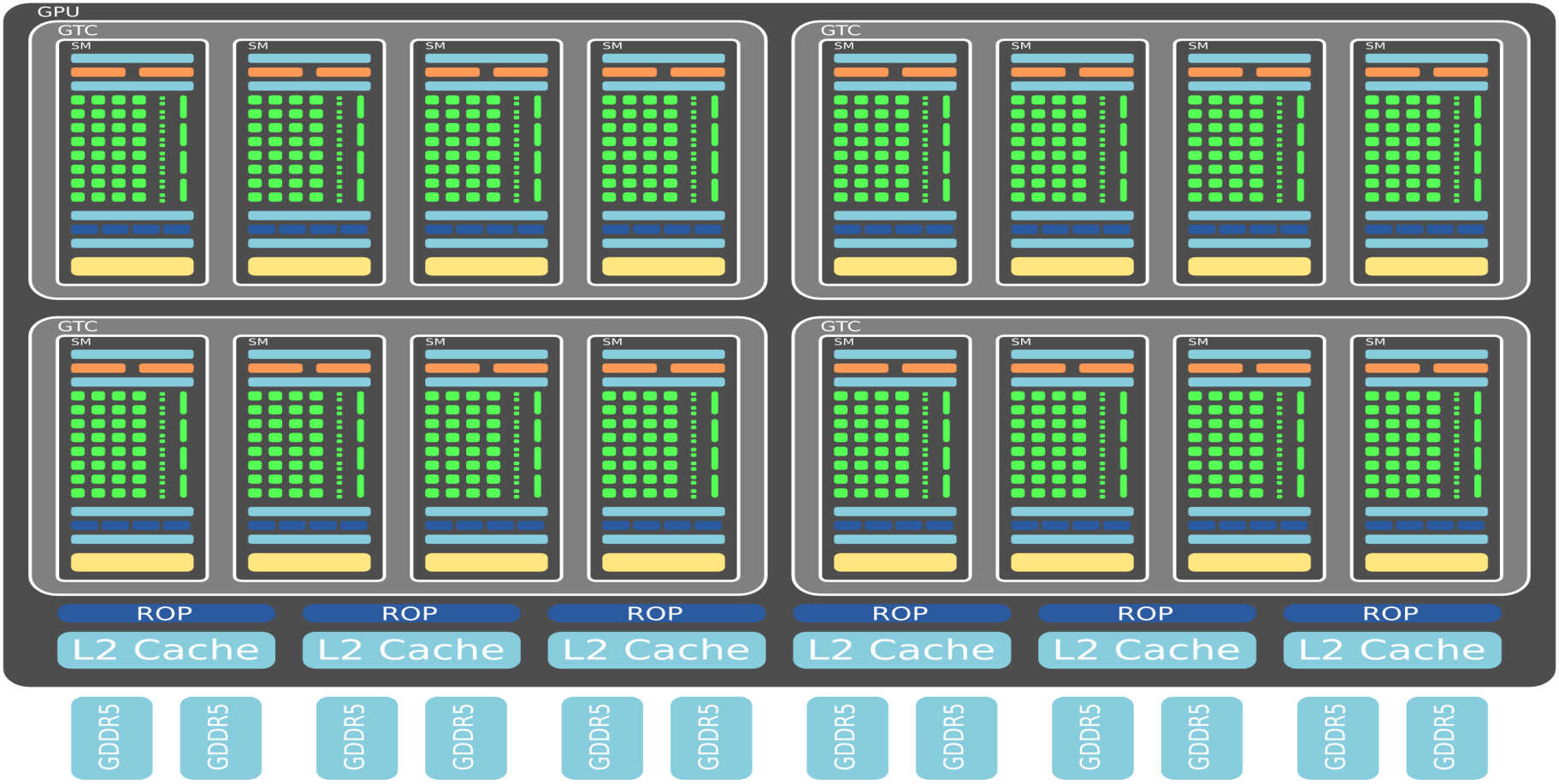
GPU

Trying it together:



GPU architecture :

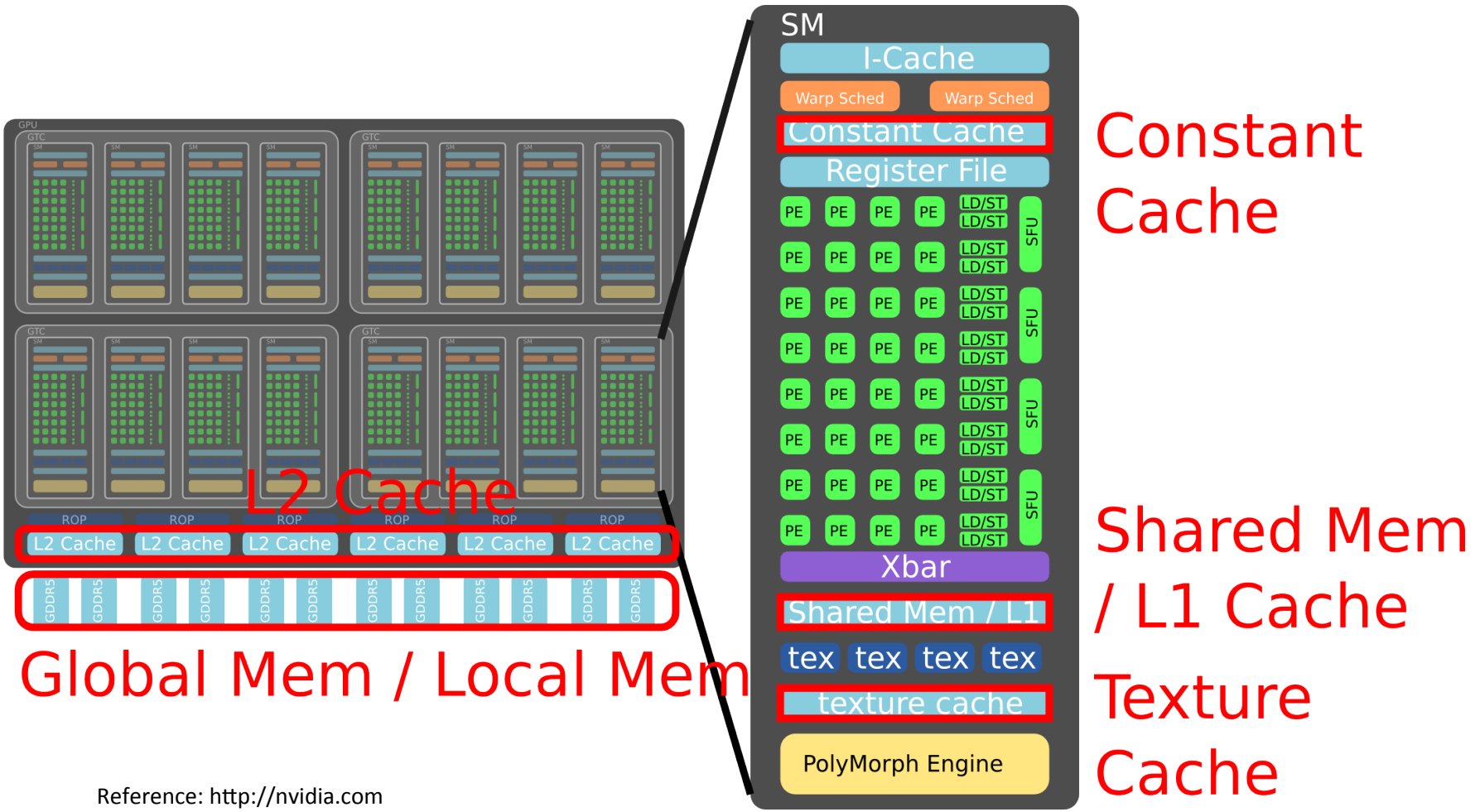
- High level block diagram of NVIDIA GPU chip.



Reference: http://nvidia.com

GPU memory model:

- Multilevel levels of memory hierarchy



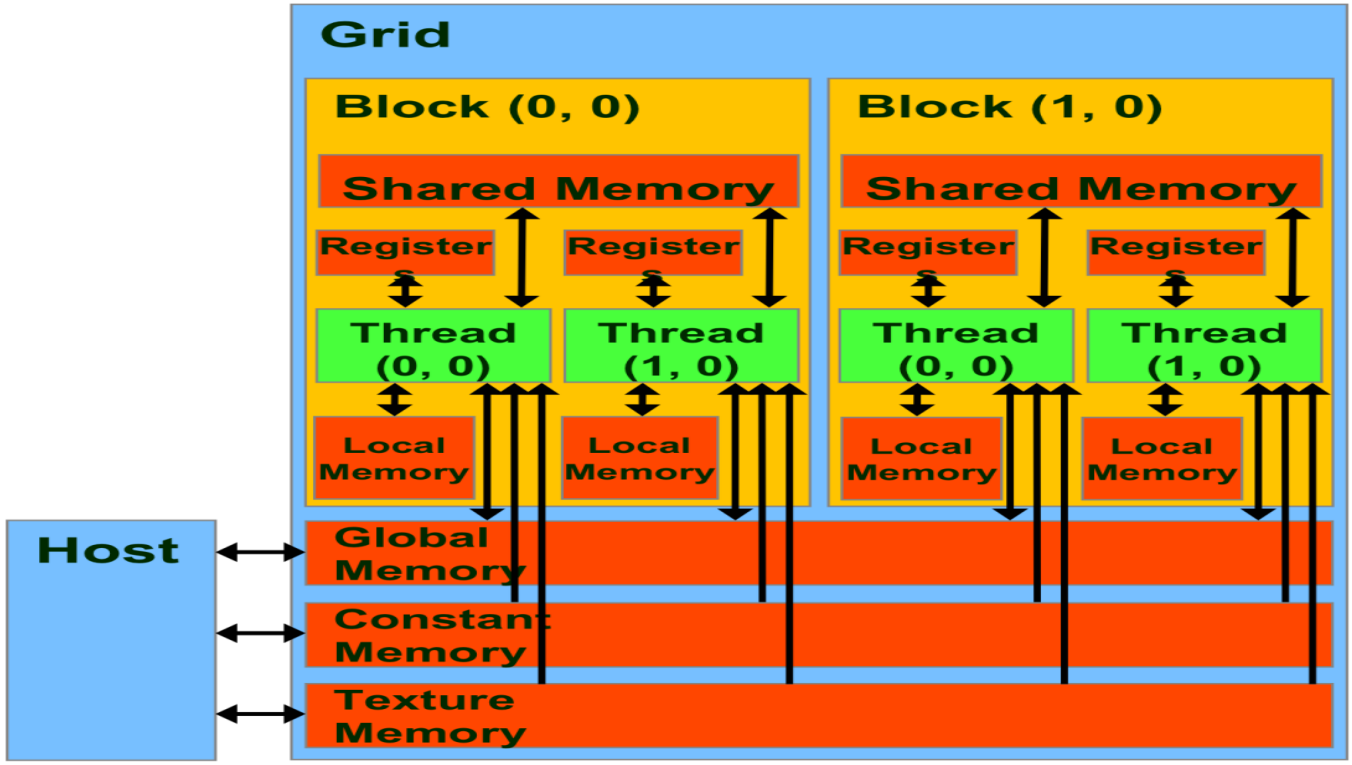
Reference: http://nvidia.com

Terminology:

- Thread: is a ready for execution/running instance of a kernel. Each thread has its own instruction address counter and register state.
- Warp: is a group of 32 parallel threads.
- Block: is a groups of Warps. A block is executed on one multiprocessor. Every block has its own shared memory and registers in the multiprocessor.
- Grid: is a group of Blocks.
- Host: is the CPU in CUDA applications.
- Device: is the GPU in CUDA applications.

GPU memory model:

- GPU has much more aggressive memory subsystem.

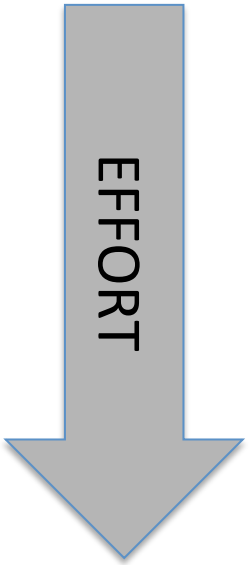


Reference: http://nvidia.com

GPU programming:

How to use GPUs

- Use existing GPU software
- Use available libraries for GPUs
- Program GPU with directives
- Program native GPU code



Use existing GPU software:

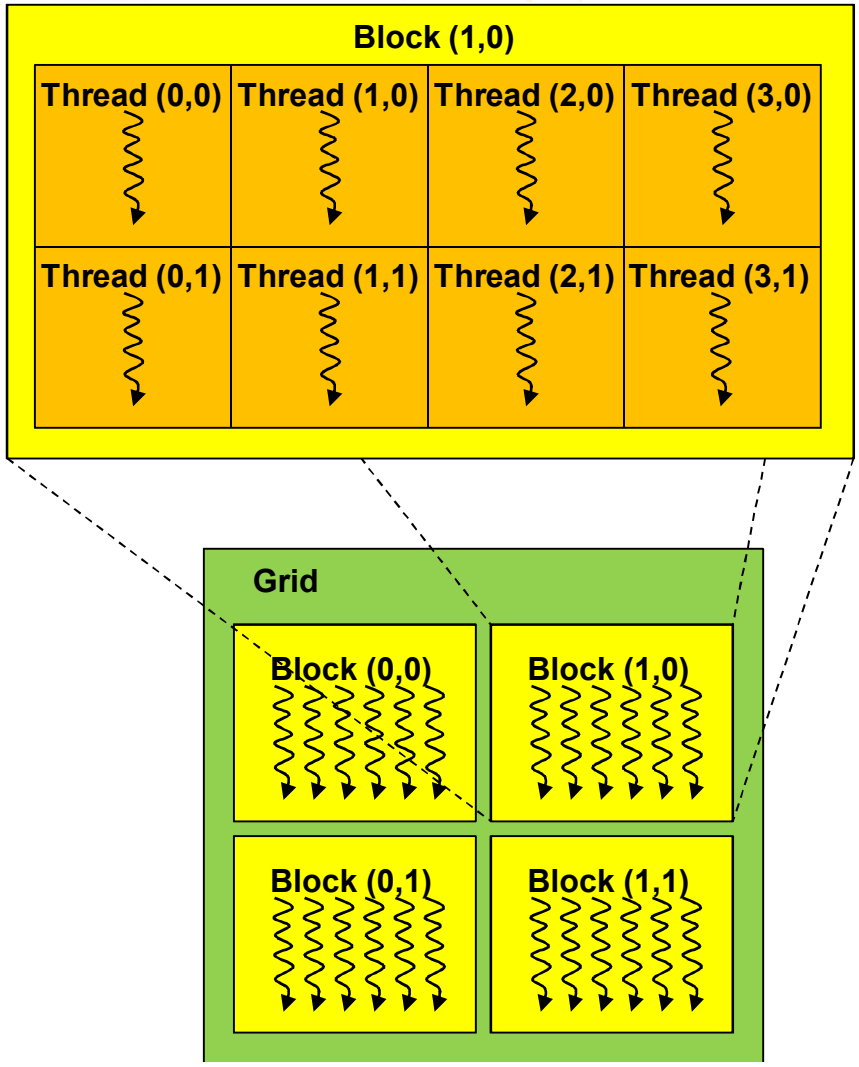
- NAMD, GROMACS, GPU-HMMER, TeraChem
- Pros
  - No implementation headaches for end users.
- Cons
  - Existing applications do not cover all science areas.
  - Often include limited number of algorithms/models.
  - For many applications the GPU version is still immature.

GPU programming model:

- GPU accelerator is called **device**, CPU is **host**.
- GPU code (kernel) is launched and executed on the device by several threads.
- Threads grouped into thread blocks.
- Program code is written from single thread's point of view.
  - Each thread can diverge and execute a unique code path (can cause performance issues)

Thread Hierarchy :

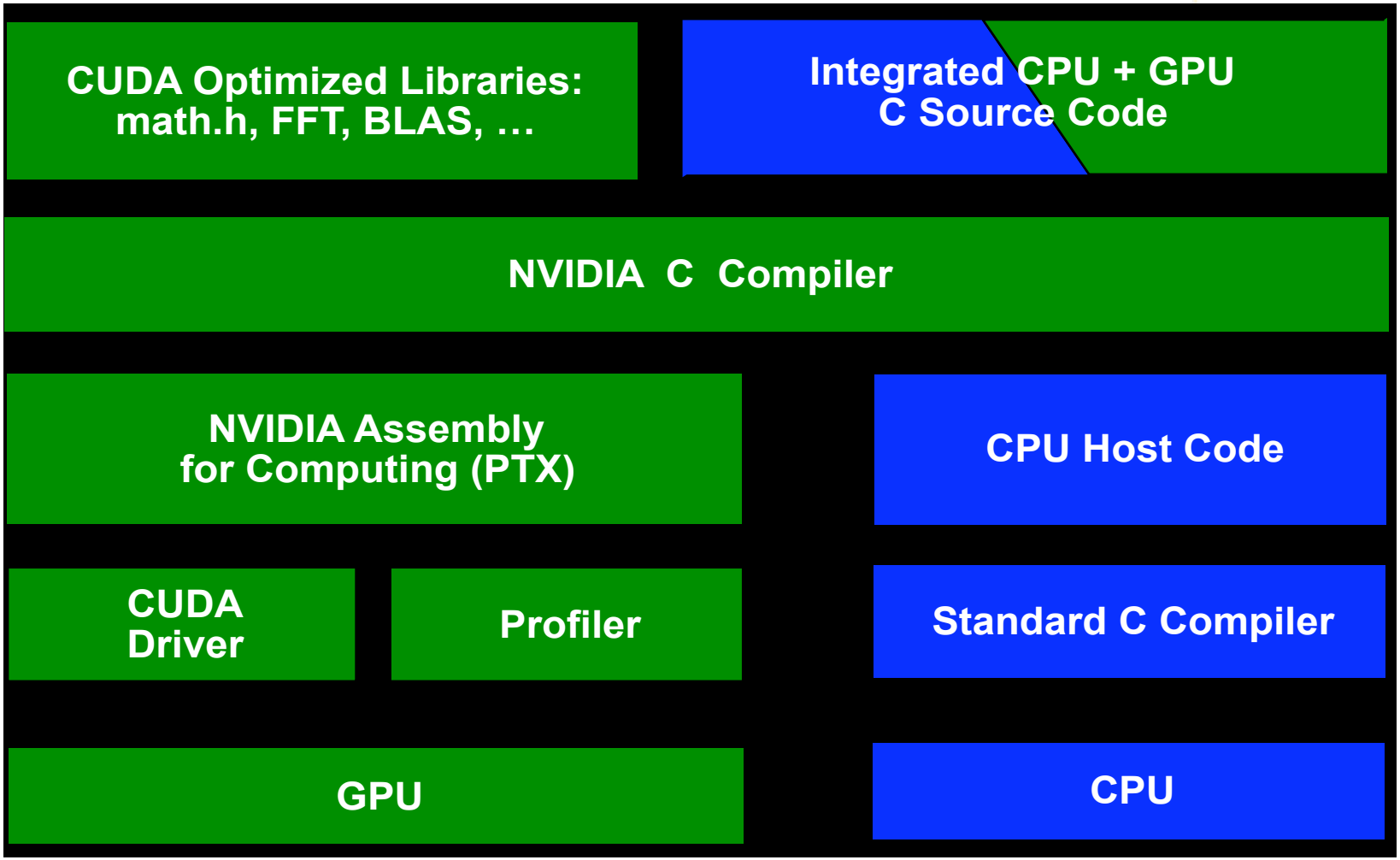
- Threads:
  - 3D IDs, unique in block
- Blocks:
  - 3D IDs, unique in grid
- Dimensions are set at kernel launch.
- Built-in variables for device code:
  - threadIdx, blockIdx
  - blockDim, gridDim



Introduction to CUDA:

- **C**ompute **U**nified **D**evice **A**rchitecture
- CUDA is a C/C++ language extension for GPU programming.
  - PGI has developed similar FORTRAN 2003 extension.
- Two APIs: Runtime and Driver

CUDA software stack:



Reference: http://nvidia.com

Introduction to CUDA C:

- Qualifiers
  - global, device, shared, local, constant…
- Built-in variables
  - threadIdx, blockIdx
- Intrinsics
  - __syncthreads,
- Runtime API
  - Memory, device execution management.
- Kernel launch

```
__device__ float array[128];
__global__ void kern(float *data)
{
__shared__ float buffer[32};
 ....
 buffer[threadIdx.x] = data[i];
 ....
 __syncthreads;
 ....
}
float *d_data;
cudaMalloc((void **))&d_data,
bytes);
kern<<<1024, 128>>>(d_data);
```

Vector add CPU:

```
void add(int *a, int *b, int *c)
{
        int i;
        for (i=0; i<N; i++)
        {
            c[i] = a[i] + b[i];
        }
}
```

CUDA memory management:

```
//include header files
#include <cuda.h>
#include <cutil.h>
//include kernels
#include "vector_add_kernel.cu"
static const int N = 100000;
int main( int argc, char** argv) {
int a[N], b[N], c[N];
int *dev_a, *dev_b, *dev_c;
//Memory allocation on device
cudaMalloc(&dev_a, N*sizeof(int);
cudaMalloc(&dev_b, N*sizeof(int);
cudaMalloc(&dev_c, N*sizeof(int);
//Memory copy host to device
cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpuKind:cudaMemcpuHostToDevice);
cudaMemcpy(dev_b, b, N*sizeof(int), cudaMemcpuKind:cudaMemcpuHostToDevice);
//Call Kernel (in next slide)
//Copy result from GPU to CPU
cudaMemcpy(c,dev_c, N*sizeof(int), cudaMemcpuKind:cudaMemcpuDeviceToHost);
//Free memory
cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);
```

Vector launch kernel:

```
add<<<1,1>>>(dev_a, dev_b, dev_c);//serial
```

```
add<<<N,1>>>(dev_a, dev_b, dev_c);//parallel
//Only the first parameter interest us right
now. The first parameter ask Cuda to execute
the function on N parallel blocks.
```

Vector add kernel function:

```
////vector_add_kernel.cu serial
static const int N=100000;
__global__ void add(int *a, int *b, int* c)
{
   for(int i=0;i<N;i++){
   c[i] = a[i] + b[i];
 }
}
```

Vector add kernel function:

```
////vector_add_kernel.cu parallel
__global__ void add(int *a, int *b, int* c)
{
      c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x]
}
```

Launching Kernels:

- Modified C function call syntax:

```
kernel<<<dim3 dG, dim3 dB>>>(...)
```

- Execution Configuration (“<<< >>>”)
- dG - dimension and size of grid in blocks Two-dimensional: x and y
  Blocks launched in the grid: dG.x * dG.y
- dB - dimension and size of blocks in threads: Three-dimensional: x, y, and z
  Threads per block: dB.x * dB.y * dB.z
- Unspecified dim3 fields initialize to 1

Compiling CUDA code:

- Compilation tools are a part of CUDA SDK.
- `nvcc` compiler translates code written in CUDA into PTX.
- `nvcc` separates the code for host and device.
  - Host code is compiled with regular C/C++ compiler.
- More information: http://www.nics.tennessee.edu/~ksharkey/tutorials/

Compiling CUDA code:

On Keeneland:

```
>module load PE-intel
>module load cuda/4.1

> nvcc -ccbin $CC -o gpu.out
gpucode.cu
```

On Nautilus:

```
>module load PE-gnu
>module load cuda/4.0RC2

> nvcc -ccbin $CC -o
gpu.out gpucode.cu
```

- More information:
  http://www.nics.tennessee.edu/~ksharkey/tutorials/

Running CUDA code:

On Keeneland:

```
> qsub -I -l nodes=1:ppn=1:gpus=3,walltime=00:30:00
> ./gpu.out
```

On Nautilus:

```
> qsub -I -l ncpus=1,gpus=1,walltime=00:30:00
> ./gpu.out
```

- More information:
  https://wiki-rdav.nics.tennessee.edu/index.php/Using_the_Nvidia_GPUs_on_Nautilus

  AND

  http://keeneland.gatech.edu/support/quick-start#runningjobs

Hybrid:

- CUDA with MultiGPU
- CUDA + OpenMP
- CUDA + MPI
- CUDA + OpenMP + MPI

CUDA libraries:

- MAGMA
- CUBLAS
- CULA
- CUFFT
- CUSPARSE
- THRUST
- Optix

- Easy to use in your programs.
- Algorithms in libraries are usually efficient.

CUDA debuggers and profilers:

- Debuggers:
  - Allinea DDT
  - CUDA-GDB
  - Totalview
  - Cuda-memcheck
- Profilers:
  - Tau
  - NVIDIA visual profiler

CUDA and OpenCL:

- NVIDIA: CUDA
- Use compiler to build kernels
- C language extensions(nvcc)
  - Also a low-level driver-only API

- Open-free standard.
- Builds kernel at runtime.
- API only, no new compiler-API calls to execute kernel

Directive based GPU code:

- Two main products
  - PGI accelerators
  - HMPP (CAPS enterprise)
- Normal C or Fortran code with directives to guide compiler in creating a GPU version.
- Backend supporting CUDA, OpenCL and even normal CPUs.

Directive based GPU code:

- Pros
  - Same code base as CPU version
  - Less time consuming
  - Portability is better due to different backends.
- Cons
  - Generated code may not be as fast as hand-tuned CUDA.

OpenACC:

- Describes a collection of compiler directives to specify loops and regions of code in standard C,C++ and Fortran.
- Allow programmer to create high-level host+ accelerators programs without the need to explicitly initialize the device, manage data or program transfers.
- Backed by PGI, CAPS, Cray and NVIDIA
- Part of OpenMP 4.0 ?
- More information: http://www.openacc-standard.org/

Ocelot:

- Aim to compile CUDA programs so that they can be run on architectures other than NIVIDA GPUs
- It is a modular dynamic compilation framework for heterogeneous system, providing various backend targets for CUDA programs and analysis modules for the PTX virtual instruction set.
- Proliferation of Heterogeneous computing.
- Ocelot currently allows CUDA programs to be executed on NVIDIA GPUs, AMD GPUs, and x86-CPUs at full speed without recompilation.

Jacket:

- It combines the speed of CUDA and the graphics of the GPU with the user friendliness.
- Provides GPU library for C, C++, Fortran, Python and MATLAB.
- Provides GPU counterparts to CPU data types, such as real and complex double, single,uint32, int32, logical, etc. Any variable residing in the host (CPU) memory can be cast to Jacket's GPU data types.
- It's memory management system allocates and manages memory for these variables on the GPU automatically, behind-the-scenes. Any functions called on GPU data will execute on the GPU automatically without any extra programming.
- For more information: http://www.accelereyes.com/

Jacket example:

CPU:

```
X = double( magic( 3 ) );
Y = ones( 3, 'double' );
A = X * Y
```

GPU:

```
addpath <jacket_root>/engine
X = gdouble( magic( 3 ) );
Y = gones( 3, 'double' );
A = X * Y
```

Programming languages and GPGPU:

- pyCUDA, pyOpenCL
- MATLAB with CUDA toolbox
- CUDA FORTRAN
- ROpenCL, RCUDA
- Haskell, Perl etc…

Resources:

- http://developer.nvidia.com/category/zone/cuda-zone
- http://gpgpu.org/
- http://developer.nvidia.com/about-parallel-forall
- http://www.gputechconf.com/page/home.html#
- http://software.intel.com/en-us/articles/vcsource-tools-opencl-sdk/
- http://developer.amd.com/pages/default.aspx
- http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CUBLAS_Library.pdf
- http://www.vpac.org/files/GPU-Slides/04.debugging_profiling_tools.pdf
- http://keeneland.gatech.edu/software/cuda
- http://developer.nvidia.com/nvidia-visual-profiler

Rules for fast GPU codes:

- Get the data on the GPU(and keep it there! If possible)
  - PCIe x16 v2.0 bus: 8GiB/s in a single direction
  - GPUs: ~180 GiB/s
- Give the GPU enough work to do
- Reuse and locate data to avoid global memory bottlenecks
- Corollary: Avoid `malloc/free`

Summary:

- Accelerated supercomputers emerging.
- GPUs offer tremendous potential to accelerate scientific applications.
- Newer generations GPUs getting easier to program.
- Challenges still remain in using them efficiently.
- Still a few cliffs:
  - HOST-GPU transfer
  - Careful memory access
  - Lots of parallelism
  - Thread divergence

Accelerated Supercomputer:

- Challenges remain
  - Applicability: Can you solve your algorithm efficiently using a GPU ?
  - Programmability: Effort of code writing that uses a GPU efficiently.
  - Portability: Incompatibilities between vendors
  - Availability: Are you able gain access to large scale system ?
  - Scalability: Can you scale the GPU software efficiently to several nodes ?

References:

- https://nimrodteam.org/meetings/team_mtg_8_10/nimrod_gpu.pdf
- http://people.maths.ox.ac.uk/~gilesm/hpc/NVIDIA/NVIDIA_CUDA_Tutorial_No_NDA_Apr08.pdf
- http://www.nvidia.com/docs/IO/105880/DS-Tesla-M-Class-Aug11.pdf
- http://www.nics.tennessee.edu/~ksharkey/tutorials/
- http://en.wikipedia.org/wiki/Graphics_processing_unit
- http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2012-02-20/08-opencl.pdf
- http://developer.amd.com/gpu_assets/OpenCL_Parallel_Computing_for_CPUs_and_GPUs_201003.pdf
- http://gamelab.epitech.eu/blogtech/?p=28
- Introduction GPU computing by Sebastian von alfthan
- Supercomputing for the Masses: Killer-Apps, Parallel Mappings, Scalability and Application Lifespan by Rob Farber
- The PTX GPU Assembly Simulator and Interpreter By N.M. Stiffler, Zheming Jin, Ibrahim Savran

Summary/Wrapping up:

- In this tutorial session, we covered
  - GPU architecture
  - GPU programming model
  - CUDA C
  - CUDA tools
  - OpenCL in brief
  - Other useful GPU tools
  - References

Thank You !!!