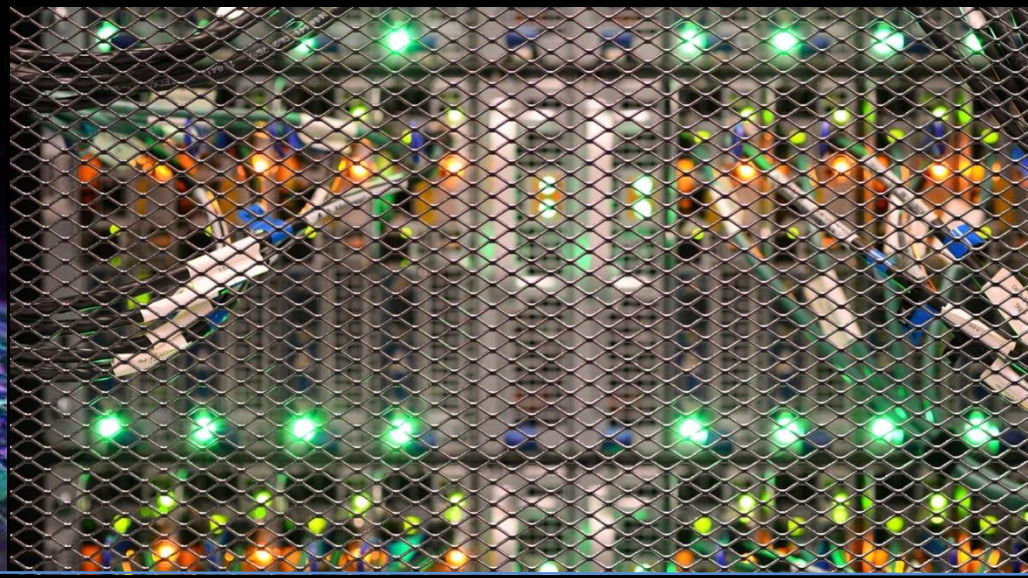


GPU Programming

using BU Shared Computing Cluster

Scientific Computing and Visualization

Boston University

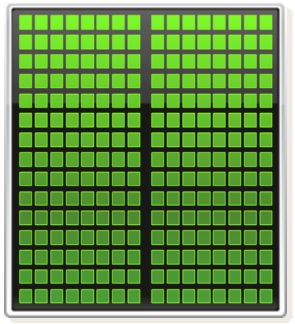
- GPU – graphics processing unit

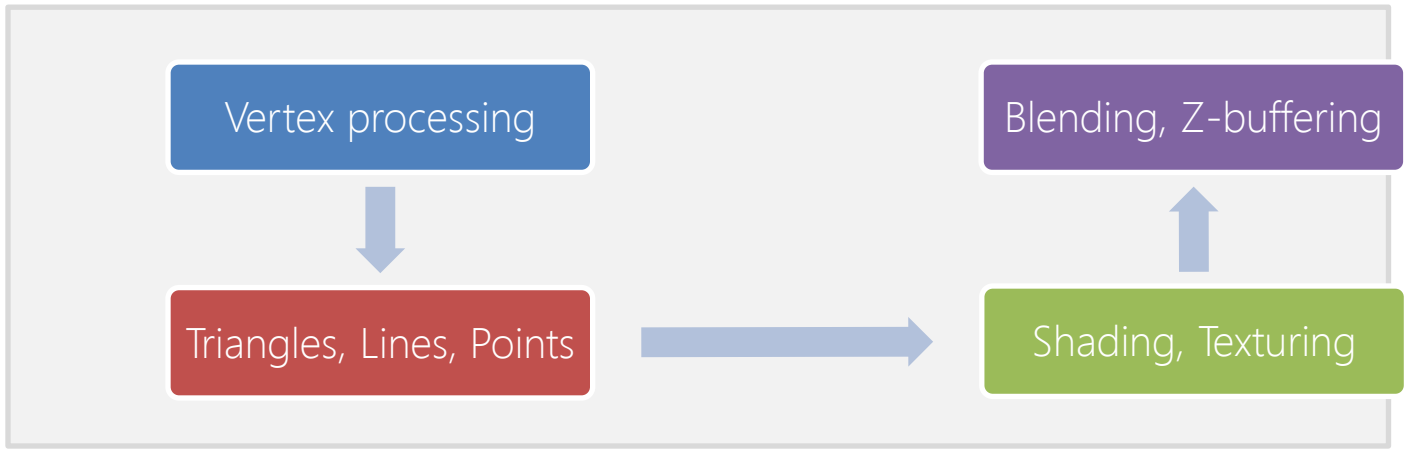
- Originally designed as a graphics processor
- Nvidia's GeForce 256 (1999) – first GPU
  - single-chip processor for mathematically-intensive tasks
  - transforms of vertices and polygons
  - lighting
  - polygon clipping
  - texture mapping
  - polygon rendering

Modern GPUs are present in

- ✓ Embedded systems
- ✓ Personal Computers
- ✓ Game consoles
- ✓ Mobile Phones
- ✓ Workstations

Traditional GPU workflow





GPGPU

1999-2000 computer scientists from various fields started using GPUs to accelerate a range of scientific applications.

GPU programming required the use of graphics APIs such as OpenGL and Cg.

2002 James Fung (University of Toronto) developed OpenVIDIA.

NVIDIA greatly invested in GPGPU movement and offered a number of options and libraries for a seamless experience for C, C++ and Fortran programmers.

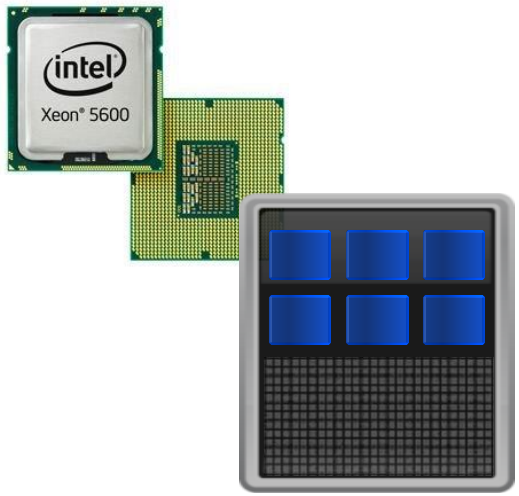
GPGPU timeline

In November 2006 Nvidia launched CUDA, an API that allows to code algorithms for execution on Geforce GPUs using C programming language.

Khronus Group defined OpenCL in 2008 supported on AMD, Nvidia and ARM platforms.

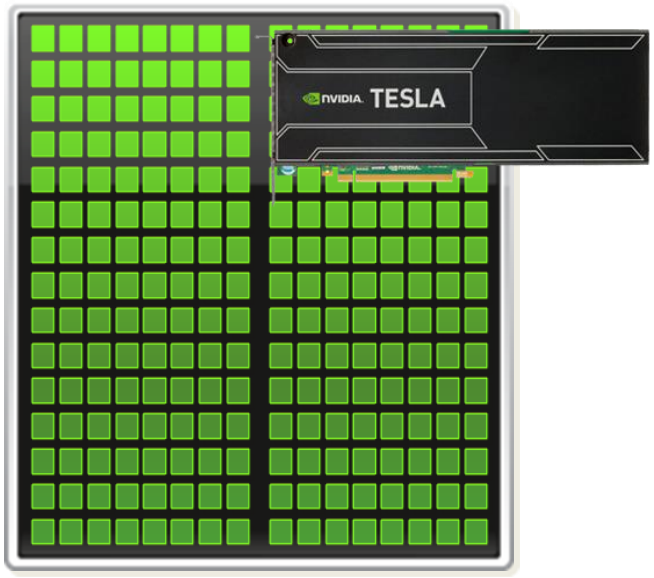
In 2012 Nvidia presented and demonstrated OpenACC - a set of directives that greatly simplify parallel programming of heterogeneous systems.

CPU



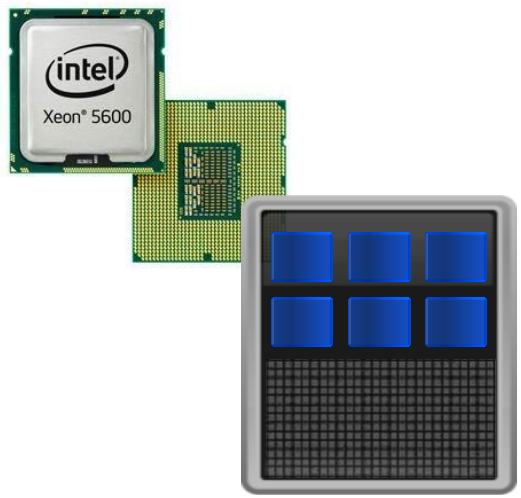
CPUs consist of a few cores optimized for serial processing

GPU

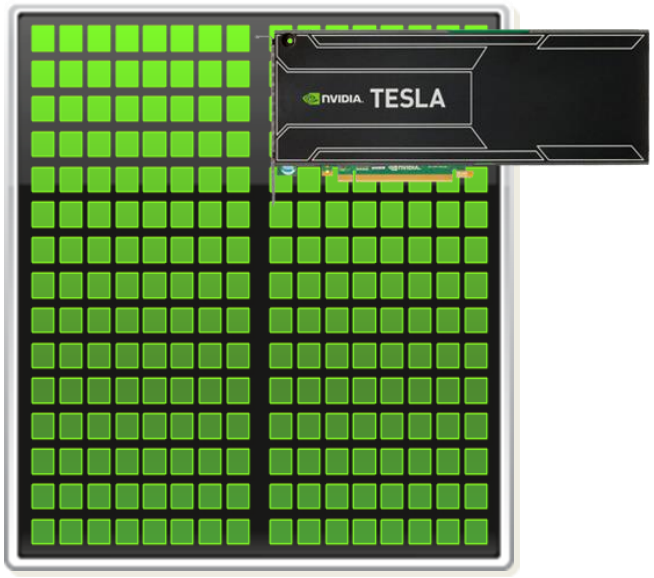


GPUs consist of hundreds or thousands of smaller, efficient cores designed for parallel performance

SCC CPU



SCC GPU



Intel Xeon X5650:

Clock speed: 2.66 GHz
4 instructions per cycle
CPU - 6 cores

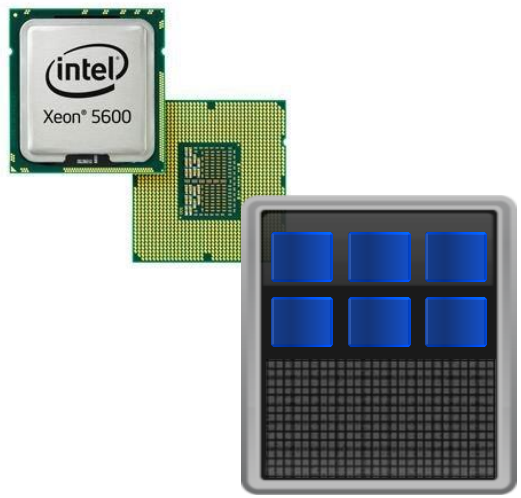
2.66 x 4 x 6 =
63.84 Gigaflops double precision

NVIDIA Tesla M2070:

Core clock: 1.15GHz
Single instruction
448 CUDA cores

1.15 x 1 x 448 =
515 Gigaflops double precision

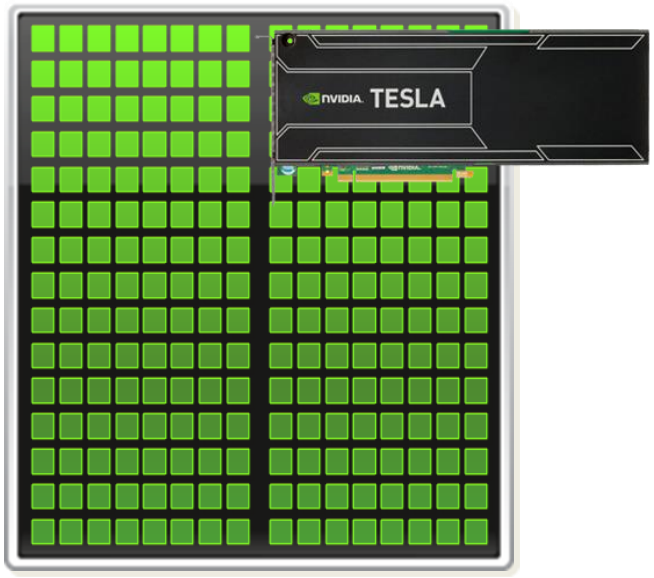
SCC CPU



Intel Xeon X5650:

Memory size: 288 GB
Bandwidth: 32 GB/sec

SCC GPU



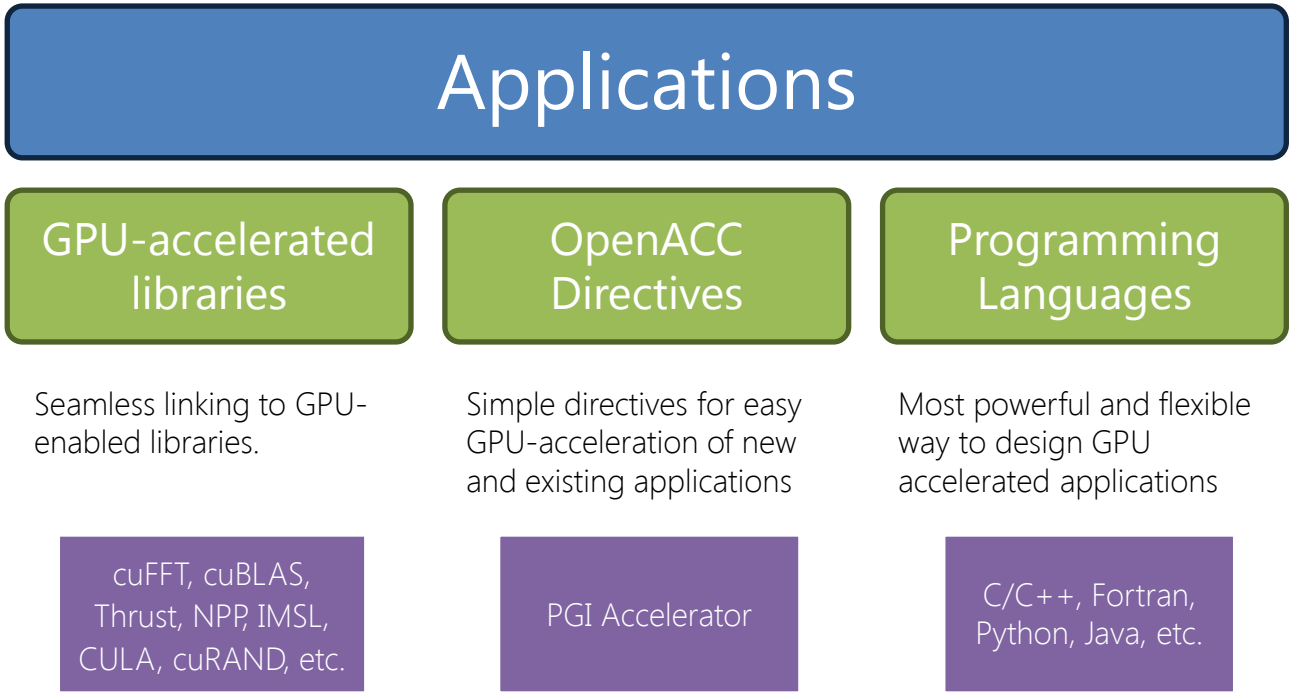
NVIDIA Tesla M2070:

Memory size: 3GB total
Bandwidth: 150 GB/sec

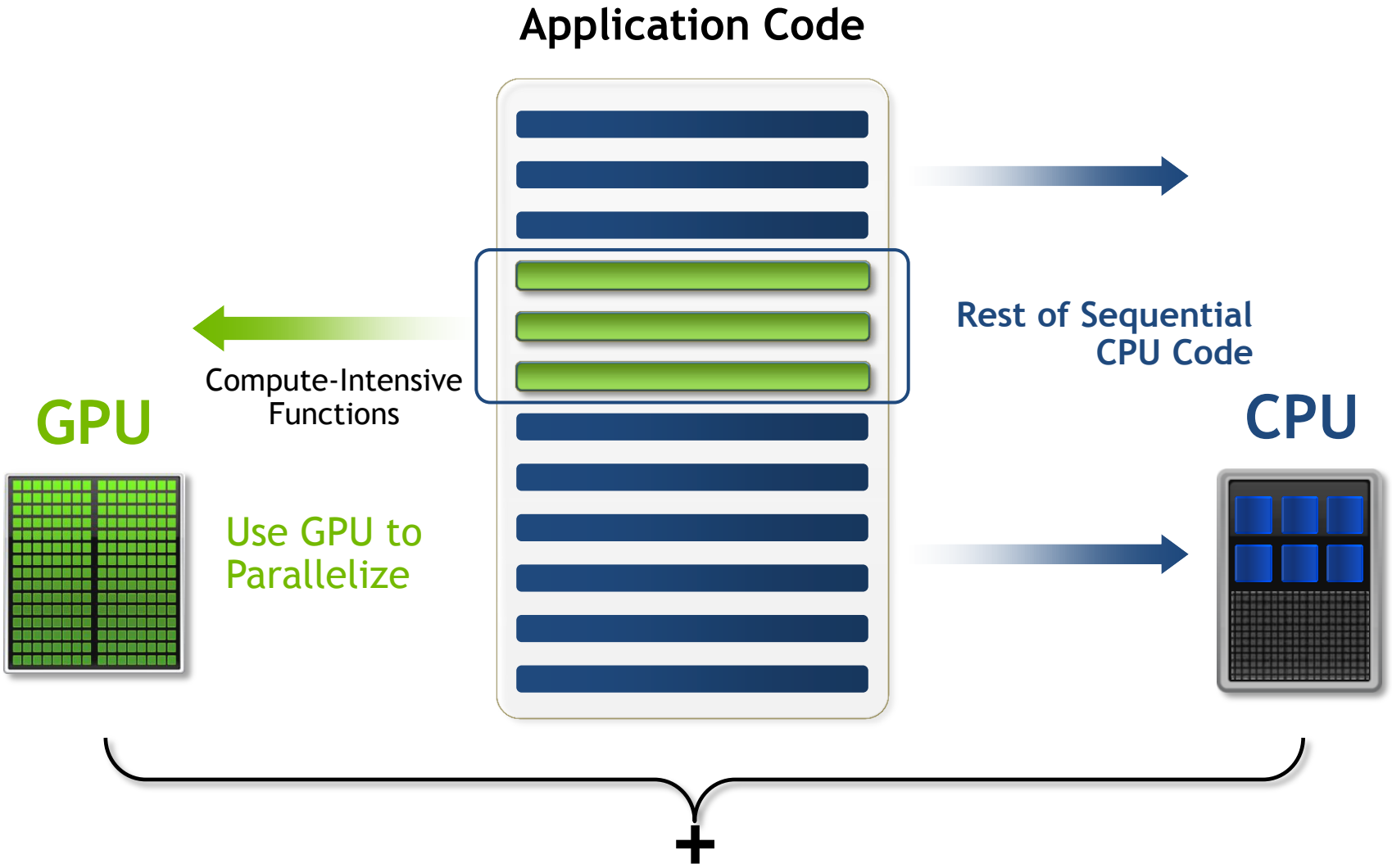
GPU Computing Growth

2008		2013
100M CUDA-capable GPUs	x 4.3	**430M** CUDA-capable GPUs
150K CUDA downloads	x 10.67	**1.6M** CUDA downloads
1 Supercomputer	x 50	**50** Supercomputers
4,000 Academic Papers	x 9.25	**37,000** Academic Papers

GPU Acceleration



Minimum Change, Big Speed-up



Will Execution on a GPU Accelerate My Application?

Computationally intensive—The time spent on computation significantly exceeds the time spent on transferring data to and from GPU memory.

Massively parallel—The computations can be broken down into hundreds or thousands of independent units of work.

C	OpenACC, CUDA
C++	Thrust, CUDA C++
Fortran	OpenACC, CUDA Fortran
Python	PyCUDA, Copperhead
Numerical analytics	MATLAB, Mathematica

GPU resources on the SCF

There are 2 sets of nodes that incorporate GPUs and are available to the SCF users:

- scc-ha1, …, scc-he2 and scc-ja1, …, scc-je2 (8 NVIDIA Tesla M2070 GPU)
- katana-k01, …, katana-k24 * (3 NVIDIA Tesla M2050 GPU)

* These nodes are part of the SCF buy-in program so access is somewhat limited to general users, based on the needs of the group who purchased this cluster. These nodes are currently part of the Katana Cluster but they will be moving to become part of the SCC during the week of July 8-12, 2013

Interactive Batch

Request xterm with access to 1 GPU for 2 hours:

```
> qsh -V  -l  h_rt=2:00:00 -l gpus=1
```

For SCC cluster add project name:

```
> qsh -V  -P <project_name> -l  h_rt=2:00:00 -l gpus=1
```

Tutorial examples

```
> cp -r  /scratch/intro_gpu  .
> cd intro_gpu
> ls -l

drwxr-xr-x 2 koleinik scv 4096 Jun 25 15:45 deviceQuery/
drwxr-xr-x 2 koleinik scv 4096 Jun 23 08:26 gemm/
drwxr-xr-x 2 koleinik scv 4096 Jun 23 08:49 gpu_matlab/
drwxr-xr-x 2 koleinik scv 4096 Jun 25 13:51 helloCuda/
-rw-r--r-- 1 koleinik scv  143 Jun 26 15:34 setpaths
drwxr-xr-x 2 koleinik scv 4096 Jun 25 15:11 vectorAdd/

// add CUDA software to the user's path
> source setpaths
```

CUDA: **Hello, World!** example

```
/* Main function, executed on host (CPU) */
int main( void) {

     /* print message from CPU */
     printf( "Hello Cuda!\n" );

     /* execute function on device (GPU) */
     hello<<<NUM_BLOCKS, BLOCK_WIDTH>>>();

     /* wait until all threads finish their job */
     cudaDeviceSynchronize();

     /* print message from CPU */
     printf( "Welcome back to CPU!\n" );

     return(0);
}
```

Kernel:
A parallel function that runs on the GPU

intro_gpu/helloCuda/helloCuda.cu

CUDA: **Hello, World!** example

```
/* Function executed on device (GPU */
__global__ void  hello( void) {

  printf( "\tHello from GPU: thread %d and block %d\n",
                                    threadIdx.x,  blockIdx.x );

}
```

intro_gpu/helloCuda/helloCuda.cu

CUDA: **Hello, World!** example

Compile and build the program using NVIDIA's **nvcc** compiler:

```
nvcc -o helloCuda helloCuda.cu -arch sm_20
```

Running the program on the GPU-enabled node:

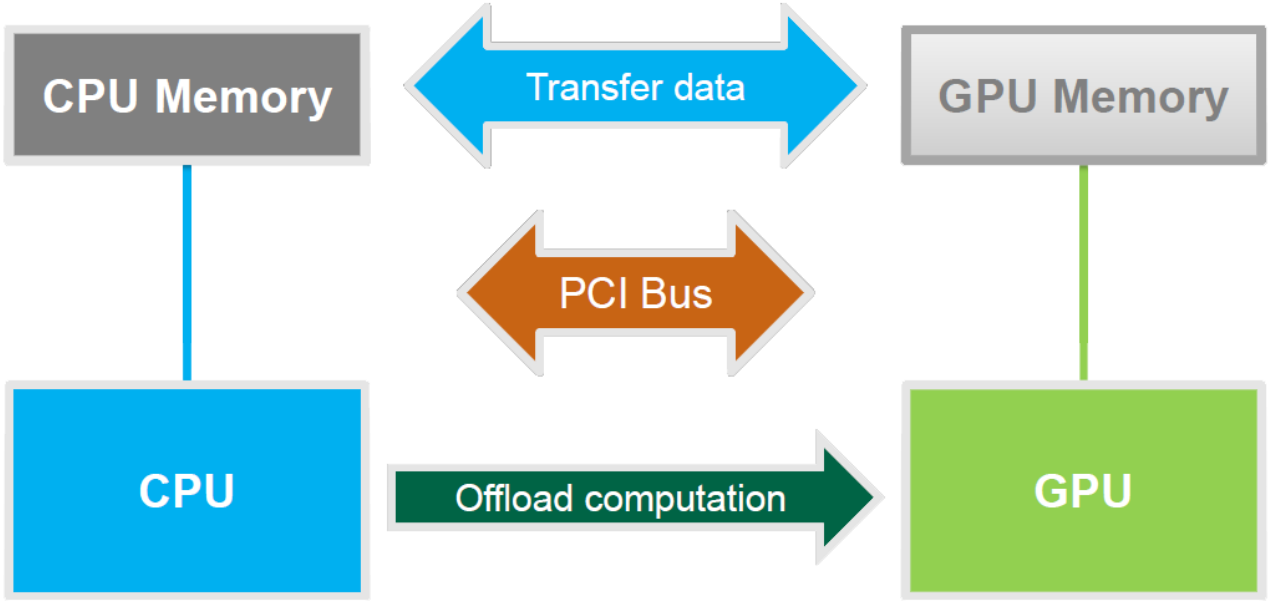
helloCuda

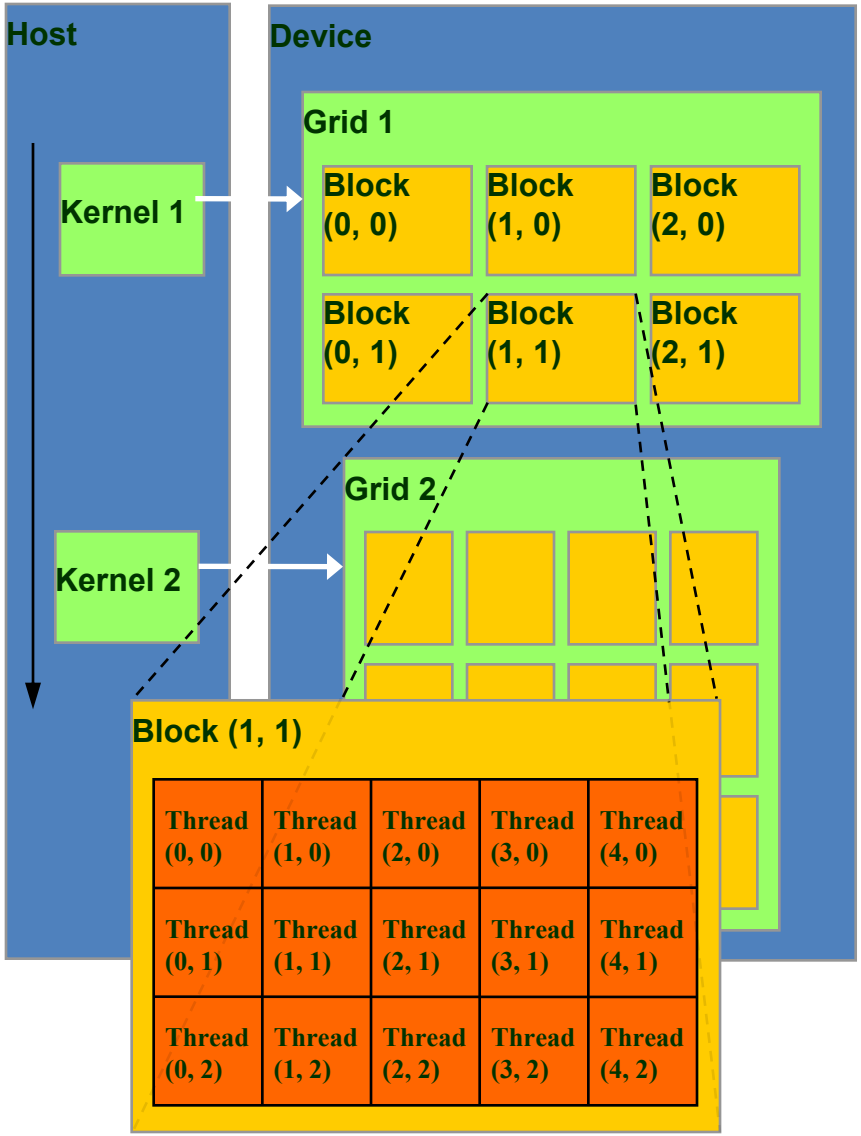
```
Hello Cuda!
        Hello from GPU: thread 0 and block 0
        Hello from GPU: thread 1 and block 0
        . . .

        Hello from GPU: thread 6 and block 2
        Hello from GPU: thread 7 and block 2
Welcome back to CPU!
```

Note:
Threads are executed on "first come, first serve" basis. Can not expect any order!

intro_gpu/helloCuda/helloCuda.cu

Basic Concepts




Host
Device
Grid 1
Kernel 1
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Grid 2
Kernel 2
Block (1, 1)
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (4, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (4, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Thread (4, 2)

CUDA: **Vector Addition** example

```
/* Main function, executed on host (CPU) */
int main( void) {


     /* 1. allocate memory on GPU */

     /* 2. Copy data from Host to GPU */

     /* 3. Execute GPU kernel */

     /* 4. Copy data from GPU back to Host */

     /* 5. Free GPU memory */

     return(0);
}
```

intro_gpu/vectorAdd/vectorAdd.cu

CUDA: **Vector Addition** example

```
/* 1. allocate memory on GPU */

float *d_A = NULL;
if (cudaMalloc((void **)&d_A, size) != cudaSuccess)
   exit(EXIT_FAILURE);

float *d_B = NULL;
cudaMalloc((void **)&d_B, size); /* For clarity we'll not check for err */

float *d_C = NULL;
cudaMalloc((void **)&d_C, size);
```

intro_gpu/vectorAdd/vectorAdd.cu

CUDA: **Vector Addition** example

```
/* 2. Copy data from Host to GPU */

cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);


cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

intro_gpu/vectorAdd/vectorAdd.cu

CUDA: **Vector Addition** example

```
/* 3. Execute GPU kernel */

/* Calculate number of blocks and threads */
int threadsPerBlock = 256;
int blocksPerGrid =(numElements + threadsPerBlock - 1) / threadsPerBlock;

/* Launch the Vector Add CUDA Kernel */
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);

/* Wait for all the threads to complete */
cudaDeviceSynchronize();
```

intro_gpu/vectorAdd/vectorAdd.cu

CUDA: **Vector Addition** example

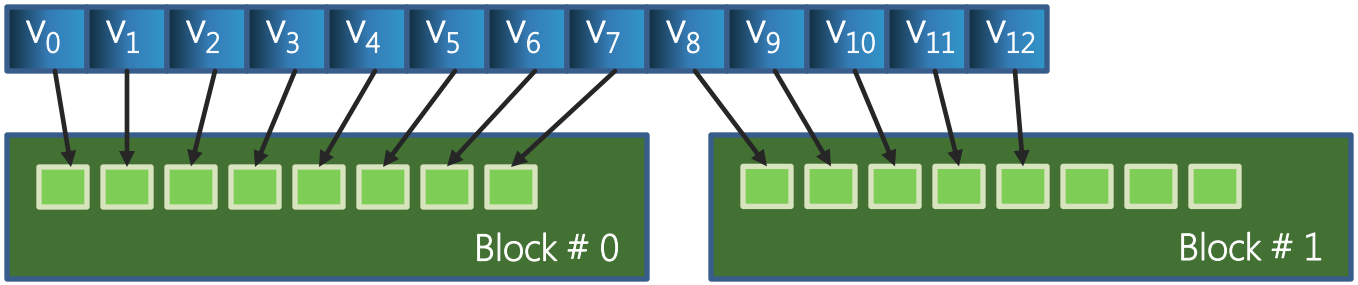
```
/* 4. Copy data from GPU back to Host */

cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

intro_gpu/vectorAdd/vectorAdd.cu

CUDA: **Vector Addition** example

```
/* 5. Free GPU memory */

cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

CUDA: **Vector Addition** example




```
/* CUDA Kernel */
__global__ void vectorAdd( const float *A,
                           const float *B,
                                 float *C,
                                 int numElements) {

      /* Calculate the position in the array */
      int i = blockDim.x * blockIdx.x + threadIdx.x;

      /* Add 2 elements of the array */
      if (i < numElements) C[i] = A[i] + B[i];
}
```

intro_gpu/vectorAdd/vectorAdd.cu

CUDA: **Vector Addition** example

```
/* To build this example, execute Makefile */
> make

/* To run, type vectorAdd: */
> vectorAdd

[Vector addition of 50000 elements]
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 196 blocks of 256 threads *
Copy output data from the CUDA device to the host memory
Done

* Note: 196 x 256 = 50176 total threads
```

intro_gpu/vectorAdd/vectorAdd.cu

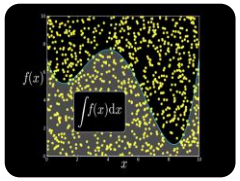
CUDA: **Device Query**

```
Device: "Tesla M2050"
  CUDA Driver Version / Runtime Version          4.2 / 4.2
  CUDA Capability Major/Minor version number:    2.0
  Total amount of global memory:                 2687 MBytes (2817982464 bytes)
  (14) Multiprocessors x ( 32) CUDA Cores/MP:    448 CUDA Cores
  GPU Clock rate:                                1147 MHz (1.15 GHz)
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  . . .
```

intro_gpu/deviceQuery/deviceQuery.cpp

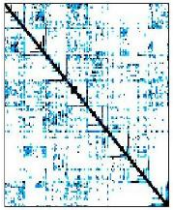
GPU Accelerated Libraries

NVIDIA cuBLAS



NVIDIA cuRAND

NVIDIA NPP



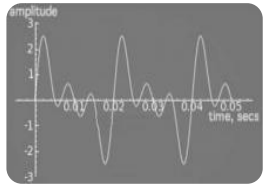
NVIDIA cuSPARSE

CUSP

Sparse Linear Algebra

Thrust

C++ STL Features for CUDA



NVIDIA cuFFT

GPU Accelerated Libraries

powerful library of parallel algorithms and data structures;

provides a flexible, high-level interface for GPU programming;

For example, the thrust::sort algorithm delivers **5**x to **100**x faster sorting performance than STL and TBB

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include <cstdlib>

int main(void)
{
  // generate random data on the host
  thrust::host_vector<int> h_vec(100);
  thrust::generate(h_vec.begin(), h_vec.end(), rand);

  // transfer to device and compute sum
  thrust::device_vector<int> d_vec = h_vec;
  int x = thrust::reduce(d_vec.begin(), d_vec.end(), 0, thrust::plus<int>());
  return 0;
}
```

GPU Accelerated Libraries

a GPU-accelerated version of the complete standard BLAS library;

6x to **17**x faster performance than the latest MKL BLAS

Complete support for all 152 standard BLAS routines

Single, double, complex, and double complex data types

Fortran binding

GEMM: $C = \alpha AB + \beta C$

```
/* General Matrix Multiply (simplified version) */
static void simple_dgemm(        int n,              double alpha,
                           const double *A,    const double *B,
                                 double beta,        double *C) {
   int i, j, k;
   for (i = 0; i < n; ++i) {
      for (j = 0; j < n; ++j){

         double prod = 0;
         for (k = 0; k < n; ++k)  prod += A[k * n + i] * B[j * n + k];

         C[j * n + i] = alpha * prod + beta * C[j * n + i];
      }
   }
}
```

intro_gpu/gemm/cuGEMM.cpp

BLAS GEMM: $C = \alpha AB + \beta C$

```
/* dgemm from BLAS library */
extern "C"{
extern void dgemm_(char *, char * ,
                   int *, int *, int *,
                   double *, double *, int *,
                   double *, int *,
                   double *, double *, int *); };

/* Main */
int main(int argc, char **argv)
{
   . . .

   /* call gemm from BLASS library */
   dgemm_("N","N", &N, &N, &N, &alpha, h_A, &N, h_B, &N, &beta, h_C_blas,&N);
   . . .
```

intro_gpu/gemm/cuGEMM.cpp

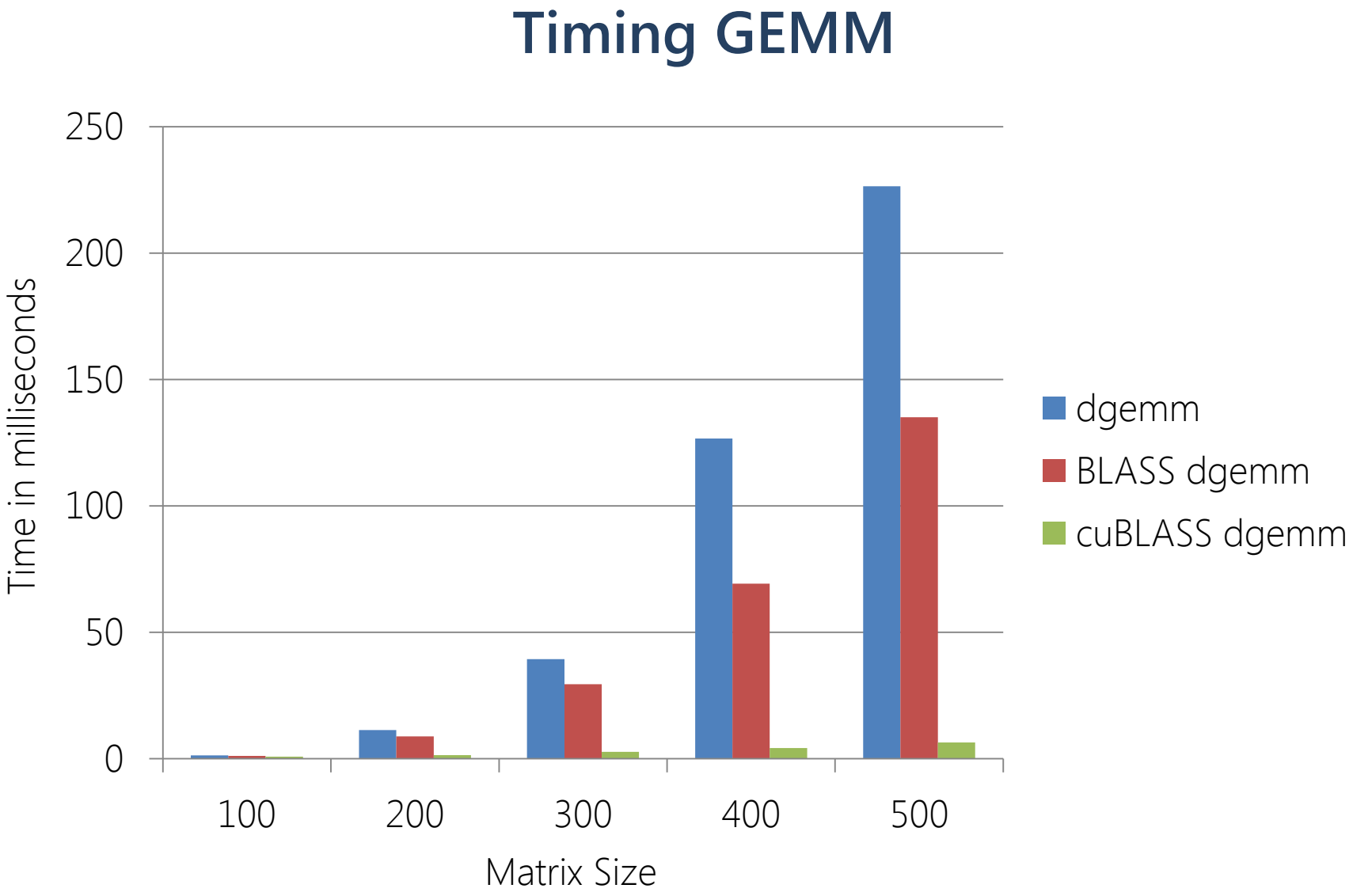
cuBLAS GEMM: $C = \alpha AB + \beta C$

```
/* Main */
int main(int argc, char **argv)
{
     /* 0. Initialize CUBLAS */
     cublasCreate(&handle);

     /* 1. allocate memory on GPU */
     cudaMalloc((void **)&d_A, n2 * sizeof(d_A[0]));

     /* 2. Copy data from Host to GPU */
     status = cublasSetVector(n2, sizeof(h_A[0]), h_A, 1, d_A, 1);

     /* 3. Execute GPU kernel */
     cublasDgemm( handle,
           CUBLAS_OP_N, CUBLAS_OP_N, N, N, N, &alpha, d_A, N, d_B, N, &beta, d_C, N );

     /* 4. Copy data from GPU back to Host */
     cublasGetVector(n2, sizeof(h_C[0]), d_C, 1, h_C, 1);

     /* 5. Free GPU memory */
     cudaFree(d_A)
}
```

intro_gpu/gemm/cuGEMM.cpp

Submitting CUDA job

```
qsub -l gpus=1 -b y cuGEMM
```


Timing GEMM
250
200
150
100
50
0
Time in milliseconds
100
200
300
400
500
Matrix Size
dgemm
BLASS dgemm
cuBLASS dgemm

Development Environment

- Nsight IDE: Linux, Mac & Windows - GPU Debugging and profiling;
- CUDA-GDB debugger (NVIDIA Visual Profiler)

CUDA Resources

- Tutorial (by SCV) is coming this fall;
- CUDA and CUDA libraries examples: http://docs.nvidia.com/cuda/cuda-samples/;
- NVIDIA's Cuda Resources: https://developer.nvidia.com/cuda-education
- Online course on Udacity: https://www.udacity.com/course/cs344
- CUDA C/C++ & Fortran: http://developer.nvidia.com/cuda-toolkit
- PyCUDA (Python): http://mathema.tician.de/software/pycuda

OpenACC Directives

- Simple compiler directives
- Works on multicore CPUs & many core GPUs
- Future integration into OpenMP

```
Program myscience

... serial code ...

!$acc compiler Directive
  do k = 1,n1
    do i = 1,n2
        ... parallel code ...
    enddo
  enddo
$acc end compiler Directive

End Program myscience
```

CPU

GPU

OpenACC Directives

- Fortran

```
!$acc directive [clause [,] clause] …]
```

Often paired with a matching end directive surrounding a structured code block

```
!$acc end directive
```

- C

```
#pragma acc directive [clause [,] clause] …]
```

Often followed by a structured code block

GEMM using OpenACC Directives

```
/* dgemm implementation with openACC acceleration*/
static void acc_dgemm( int n, double alpha, const double *A,
                       const double *B, double beta, double *C) {

   int i, j, k;

   #pragma acc parallel loop copyin(A[0:(n*n)], B[0:(n*n)]) copy(C[0:(n*n)])
   for (i = 0; i < n; ++i) {

      #pragma acc loop
      for (j = 0; j < n; ++j){

         double prod = 0;
         for (k = 0; k < n; ++k) prod += A[k * n + i] * B[j * n + k];


         C[j * n + i] = alpha * prod + beta * C[j * n + i];
      }
   }
}
```

intro_gpu/gemm/accGEMM.c

Building OpenACC program

- C:

```
pgcc -acc -Minfo -o accGEMM accGEMM.c
```

- Fortran:

```
pgfortran -acc -Minfo -o accGEMM accGEMM.f90
```

```
pgaccelinfo  /* check NVIDIA GPU and CUDA drivers */
```

-acc turns on the OpenACC feature
-Minfo returns additional information on the compilation

Current system default version of PGI compiler (8.0) does not support OpenACC.
The newest version is accessible at
/usr/local/apps/pgi-13.2/linux86-64/13.2/bin

PGI compiler output:

```
acc_dgemm:
     34, Generating present_or_copyin(B[0:n*n])
         Generating present_or_copyin(A[0:n*n])
         Generating present_or_copy(C[0:n*n])
         Accelerator kernel generated
         35, #pragma acc loop gang /* blockIdx.x */
         41, #pragma acc loop vector(256) /* threadIdx.x */
     34, Generating NVIDIA code
         Generating compute capability 1.3 binary
         Generating compute capability 2.0 binary
         Generating compute capability 3.0 binary
     38, Loop is parallelizable
     41, Loop is parallelizable
```

OpenACC Resources

- Tutorial (by SCV) is coming this fall;
- Information, examples, FAQ: http://openacc.org;
- NVIDIA's OpenACC Resources: https://developer.nvidia.com/cuda-education

MATLAB with GPU-acceleration

Use GPUs with MATLAB through **Parallel Computing Toolbox**

- GPU-enabled MATLAB functions such as fft, filter, and several linear algebra operations
- GPU-enabled functions in toolboxes: Communications System Toolbox, Neural Network Toolbox, Phased Array Systems Toolbox and Signal Processing Toolbox
- CUDA kernel integration in MATLAB applications, using only a single line of MATLAB code

```
A=rand(2^16,1);

B=fft(A);
```

```
A=gpuArray(rand(2^16,1));

B=fft(A);
```

Simple MATLAB example

```
Ga = gpuArray(rand(1000, 'single'));
Gfft = fft(Ga);
Gb = (real(Gfft) + Ga) * 6;
G = gather(Gb);
```

intro_gpu/gpu_matlab/gpuSimple.m

Matrix Product it MATLAB using GPU

```
% matrix product on Client (CPU)
C = A*B;

% copy A and B from Client to GPU
a = gpuArray(A); b = gpuArray(B);

% matrix product on GPU
c = a*b;

% copy data from GPU to Client
CC = gather(c);
```

intro_gpu/gpu_matlab/gpuExample.m

Submitting GPU MATLAB job

```
#!/bin/csh
#
# Set the hard runtime (aka wallclock) limit for this job
#$ -l h_rt=2:00:00
#
# Merge stderr into the stdout file, to reduce clutter.
#$ -j y
#
# Specifies number of GPUs wanted
#$ -l gpus=1
#
matlab -nodisplay -singleCompThread -r  \
       "N=3000; gpuExample(rand(N),rand(N)); exit"

# end of script
```

intro_gpu/gpu_matlab/matlab_batch

Running CUDA code in MATLAB

starting R2013a (available on SCC cluster only)

Example 1:

```
// cuda-kernel: add 2 numbers
__global__ void addnums (double *pi, double c){

  *pi += c;
}
```

Example 2:

```
// cuda-kernel: add 2 vectors
__global__ void addvecs (double *v1, double *v2){

  int idx = threadIdx.x;
    v1[idx] += v2[idx];
}
```

intro_gpu/gpu_matlab/add.cu

Compiling and running CUDA MATLAB code

Example 1:

1.At the command prompt type(to create ptx file for matlab):

```
nvcc -ptx add.cu  //at SCC prompt
```

2.To specify the entry point for **MATLAB** kernel,run(at matlab prompt):

```
k = parallel.gpu.CUDAKernel('add.ptx', 'addnums.cu'); //in matlab
```

3. Run kernel (kernel takes 2 arguments):

```
out = feval(k, 7, 21); //in matlab
```

intro_gpu/gpu_matlab/add.cu

Compiling and running CUDA MATLAB code

Example 2:

1.At the command prompt type(to create ptx file for matlab):

```
nvcc -ptx add.cu  //at SCC prompt
```

2.To specify the entry point for **MATLAB** kernel,run(at matlab prompt):

```
k = parallel.gpu.CUDAKernel('add.ptx', 'addvecs.cu'); //in matlab
```

3. Run kernel (kernel takes 2 arguments):

```
N = 128;
k.ThreadBlockSize = N;
feval(k, ones(N, 1), ones(N, 1));
```

intro_gpu/gpu_matlab/add.cu

MATLAB GPU Resources

- MATLAB GPU Computing Support for NVIDIA CUDA-Enabled GPUs:

http://www.mathworks.com/discovery/matlab-gpu.html;

- GPU-enabled functions :

http://www.mathworks.com/help/distcomp/using-gpuarray.html#bsloua3-1

- GPU-enabled functions in toolboxes:

http://www.mathworks.com/products/parallel-computing/builtin-parallel-support.html

This tutorial has been made possible by
Scientific Computing and Visualization
group
at Boston University.

Katia Oleinik
koleinik@bu.edu

http://www.bu.edu/tech/research/training/tutorials/list/