# Programming GPUs with CUDA

Pawel Pomorski, *HPC Software Analyst*
SHARCNET, University of Waterloo
**ppomorsk@sharcnet.ca**

# #1 system on Fall 2012 TOP500 list - Titan


Oak Ridge National Labs - operational in October 2012

18,688 nodes, each node:

Opteron 16-core CPU

NVIDIA Tesla K20 **GPU**

1.4 tera FLOPS per node

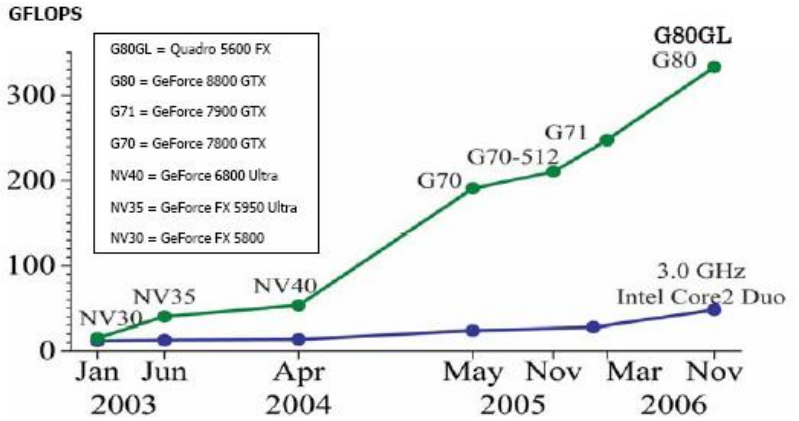
17.6 peta FLOPS total

# #1 system on Fall 2018 TOP500 list - Summit


Oak Ridge National Labs - operational in June 2018
4600 nodes, each node:
2 IBM POWER9 22-core CPUs
6 Nvidia Volta GV100 **GPUs**
40 tera FLOPS per node
200 peta FLOPS total

# Graham cluster

# GPU computing timeline




before 2003 - Calculations on GPU, using graphics API
2003 - Brook “C with streams”
2005 - Steady increase in CPU clock speed comes to a halt, switch to multicore chips to compensate. At the same time, computational power of GPUs increases
November, 2006 - CUDA released by NVIDIA
November, 2006 - CTM (Close to Metal) from ATI
December 2007 - Succeeded by AMD Stream SDK
December, 2008 - Technical specification for OpenCL1.0 released
April, 2009 - First OpenCL 1.0 GPU drivers released by NVIDIA
August, 2009 - Mac OS X 10.6 Snow Leopard released, with OpenCL 1.0 included
September 2009 - Public release of OpenCL by NVIDIA
December 2009 - AMD release of ATI Stream SDK 2.0 with OpenCL support
March 2010 - CUDA 3.0 released, incorporating OpenCL
May 2011 - CUDA 4.0 released, better multi-GPU support
mid-2012 - CUDA 5.0
late-2012 - NVIDIA K20 Kepler cards
Future - CPUs will have so many cores they will start to be treated as GPUs?
Accelerators become universal?

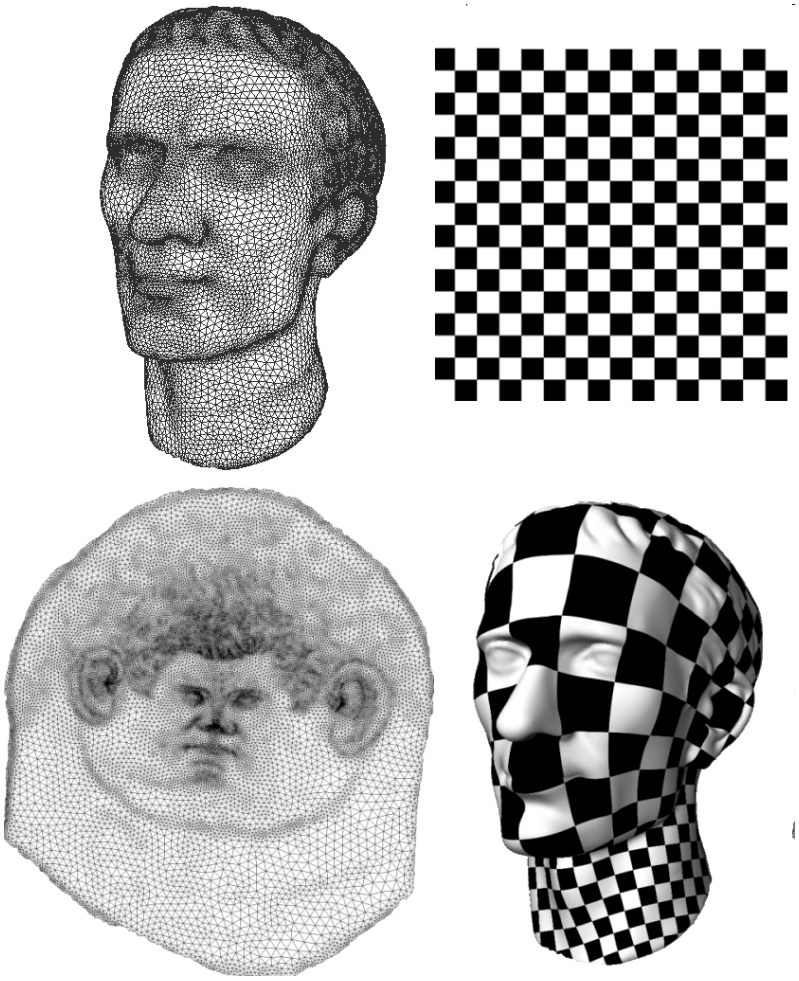
# Introduction to GPU programming

- A graphics processing unit (GPU) is a processor whose main job is to accelerate the rendering of 3D graphics primitives. Performance gains were mostly high performance computer gaming market

- GPU makers have realized that with relatively little additional silicon a GPU can be made into a general purpose computer. They have added this functionality to increase the appeal of cards.
- Even computer games now increasingly take advantage of general compute for game physics simulation

# A brief tour of graphics programming

- 2D textures are wrapped around 3D meshes to assign colour to individual pixels on screen
- Lighting and shadow are applied to bring out 3D features
- Shaders allow programmers to define custom shadow and lighting techniques
  - can also combine multiple textures in interesting ways
- Resulting pixels get sent to a frame buffer for display on the monitor



# General computing APIs for GPUs

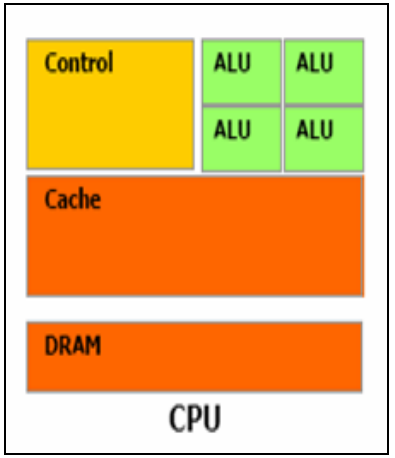
- NVIDIA offers ***CUDA*** while AMD has offered ***OpenCL*** (also supported by NVIDIA)
- More recently AMD offers a clone of CUDA via ROCm toolkit
- These computing platforms bypass the graphics pipeline and expose the raw computational capabilities of the hardware. Programmer needs to know nothing about graphics programming.
- **OpenACC** compiler directive approach is emerging as an alternative (works somewhat like OpenMP, which is also available)
- Alternative to CUDA: ***OpenCL***
  - a vendor-agnostic computing platform
  - supports vendor-specific extensions akin to OpenGL
  - goal is to support a range of hardware architectures, including GPUs, CPUs, using a standard low-level API

# The appeal of GPGPU

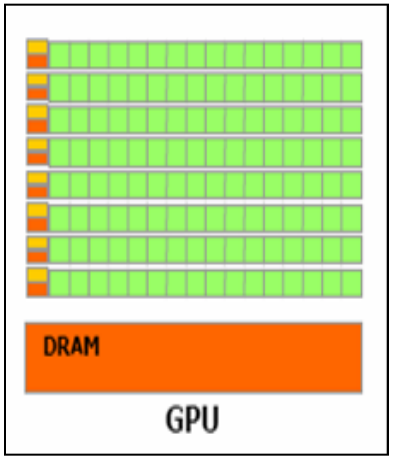
- "Supercomputing for the masses"
  - significant computational horsepower at an attractive price point
  - readily accessible hardware
- Scalability
  - programs can execute without modification on a run-of-the-mill PC with a $150 graphics card or a dedicated multi-card supercomputer worth thousands of dollars
- Bright future – the computational capability of GPUs doubles each year
  - more thread processors, faster clocks, faster DRAM, …
  - "GPUs are getting faster, faster"

# Comparing GPUs and CPUs

- CPU
  - "Jack of all trades"
  - task parallelism (diverse tasks)
  - minimize latency
  - multithreaded
  - some SIMD

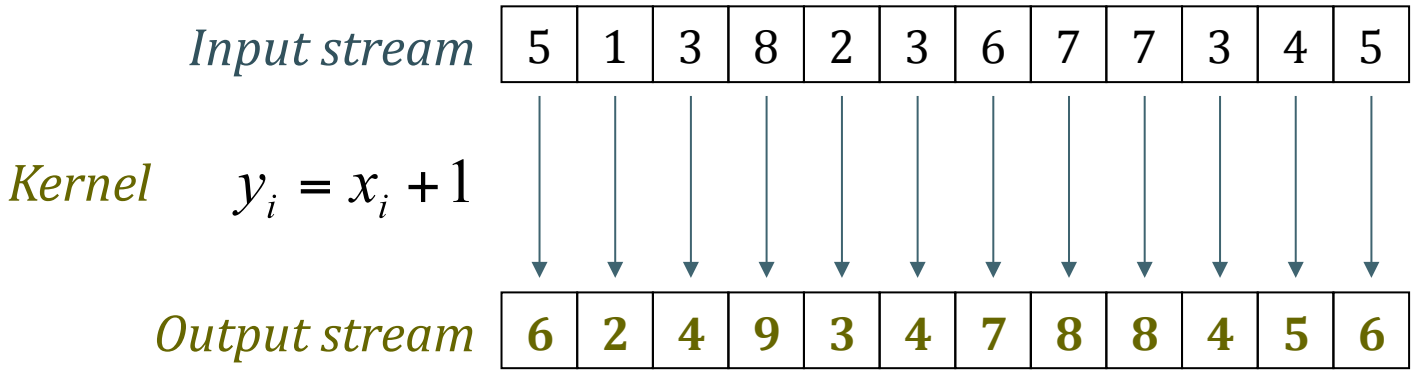



- GPU
  - excel at number crunching
  - data parallelism (single task)
  - maximize throughput
  - super-threaded
  - large-scale SIMD



# Stream computing

- A parallel processing model where a computational ***kernel*** is applied to a set of data (a ***stream***)
  - the kernel is applied to stream elements in parallel




- GPUs excel at this thanks to a large number of processing units and a parallel architecture

# Beyond stream computing

- Current GPUs offer functionality that goes beyond mere stream computing
- Shared memory and thread synchronization primitives eliminate the need for data independence
- Gather and scatter operations allow kernels to read and write data at arbitrary locations

# CUDA

- "Compute Unified Device Architecture
- A platform that exposes NVIDIA GPUs as general purpose *compute devices*
- Is CUDA considered GPGPU?
  - yes and no
    - CUDA can execute on devices with no graphics output capabilities (the NVIDIA Tesla product line) – these are not "GPUs", per se
    - however, if you are using CUDA to run some generic algorithms on your graphics card, you are indeed performing some ***G***eneral ***P***urpose computation on your ***G***raphics ***P***rocessing ***U***nit…


# Speedup

- What kind of speedup can I expect?
  - 0x – 2000x reported
  - 10x – 80x considered typical
  - >= 30x considered worthwhile (if coding required)
  - **maximum 5-10x speedup for fully optimized libraries**
- Speedup depends on
  - problem structure
    - need many identical independent calculations
    - preferably sequential memory access
  - level of intimacy with hardware
  - time investment

# How to get running on the GPU?

- Easiest case: the package you are using already has a GPU-accelerated version. No programming needed.
- Medium case: your program spends most of its time in library routines which have GPU accelerated versions. Use libraries that take advantage of GPU acceleration. Small programming effort required.
- Hard case: You cannot take advantage of the easier two possibilities, so you must convert some of your code to CUDA or OpenCL
- Newly available OpenACC framework is an alternative that should make coding easier.

# GPU-enabled software

- A growing number of popular scientific software packages have now been accelerated for the GPU
- Using a GPU accelerated package requires no programming effort for the user
- Acceleration of Molecular Dynamics software has been particularly successful, with all major packages offering the GPU acceleration option
- Many of these software packages use a combination of GPUs and CPUs in parallel. The resources used must be carefully chosen so that the program is running efficiently.
- Performance may depend on problem size.

# GPU applications

- The GPU can be utilized in different capacities

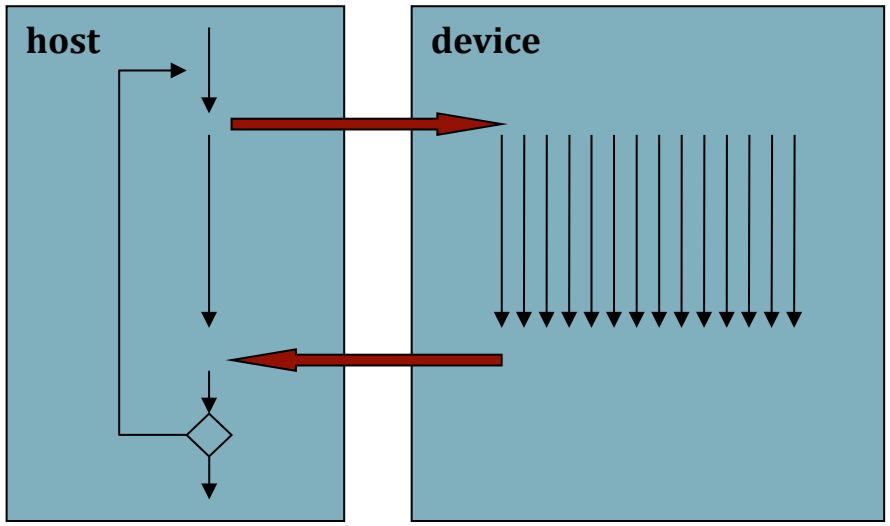
- One is to use the GPU as a massively parallel coprocessor for number crunching applications
  - upload data and kernel to GPU
  - execute kernel
  - download results
  - CPU and GPU can execute asynchronously

- Some applications use the GPU for both data crunching and visualization
  - CUDA has bindings for OpenGL and Direct3D

# CUDA programming model

- The main CPU is referred to as the ***host***
- The compute device is viewed as a ***coprocessor*** capable of executing a large number of lightweight threads in parallel
- Computation on the device is performed by ***kernels***, functions executed in parallel on each data element
- Both the host and the device have their own ***memory***
  - the host and device cannot directly access each other's memory, but data can be transferred using the runtime API
- The host manages all memory allocations on the device, data transfers, and the invocation of kernels on the device

# GPU as coprocessor

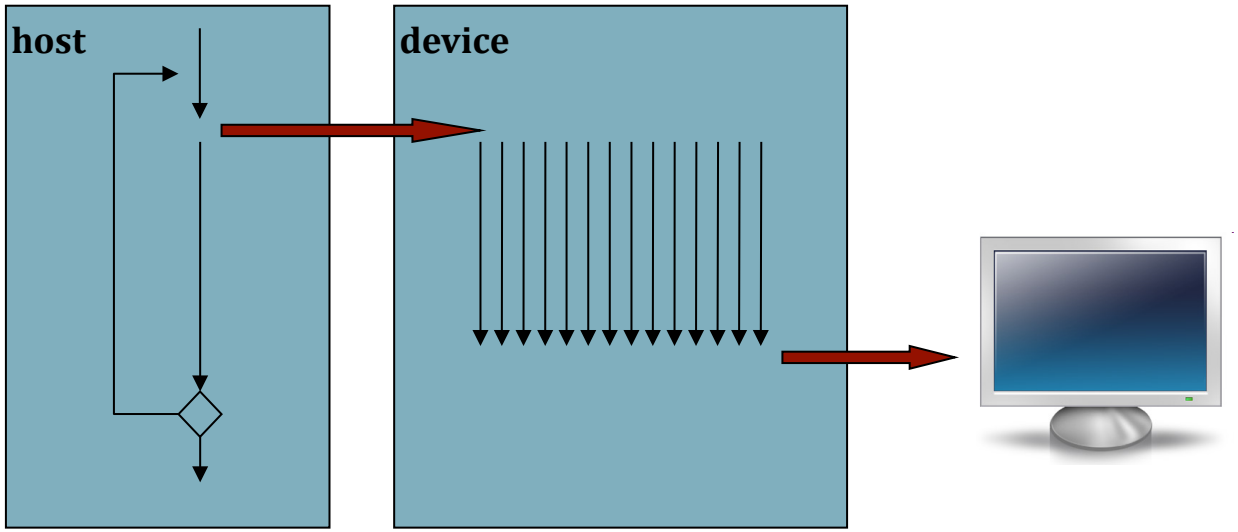



Kernel execution is asynchronous

Asynchronous memory transfers also available

- Basic paradigm
  - host uploads inputs to device
  - host remains busy while device performs computation
    - prepare next batch of data, process previous results, etc.
  - host downloads results
- Can be iterative or multi-stage

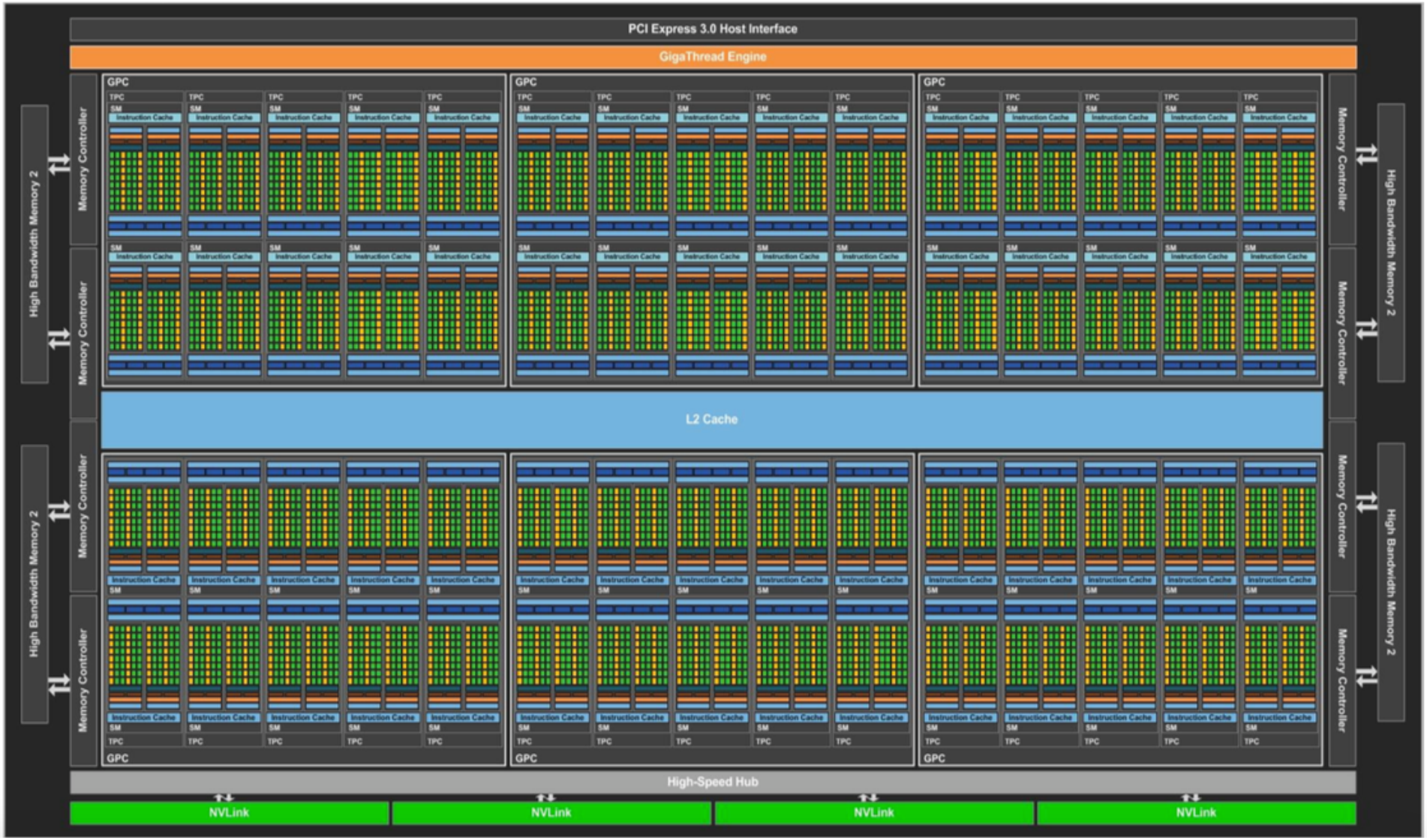
# Simulation + visualization




- Basic paradigm
  - host uploads inputs to device
  - host may remain busy while device performs computation
    - prepare next batch of data, etc.
  - results used on device for rendering, no download to host

```
[ppomorski@mcs1 deviceQuery]$ ./deviceQuery
Detected 1 CUDA Capable device(s)
Device 0: "Tesla P100-PCIE-16GB"
  CUDA Driver Version / Runtime Version          10.1 / 10.0
  CUDA Capability Major/Minor version number:    6.0
  Total amount of global memory:                 16281 MBytes (17071734784 bytes)
  (56) Multiprocessors, ( 64) CUDA Cores/MP:     3584 CUDA Cores
  GPU Max Clock rate:                            1329 MHz (1.33 GHz)
  Memory Clock rate:                             715 Mhz
  Memory Bus Width:                              4096-bit
  L2 Cache Size:                                 4194304 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384,
16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 59 / 0
  Compute Mode:
  < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

# GPU Hardware architecture - NVIDIA Pascal



# GPU Hardware architecture - NVIDIA Pascal

# Hardware basics

- The compute device is composed of a number of ***multiprocessors***, each of which contains a number of SIMD processors
  - Tesla P100 has 56 multiprocessors (each with 64 SP CUDA cores and 32 DP CUDA cores)

- A multiprocessor can execute K ***threads*** in parallel physically, where K is called the ***warp size***
  - ***thread*** = instance of kernel
  - warp size on current hardware is 32 threads

- Each multiprocessor contains a large number of 32-bit ***registers*** which are divided among the active threads

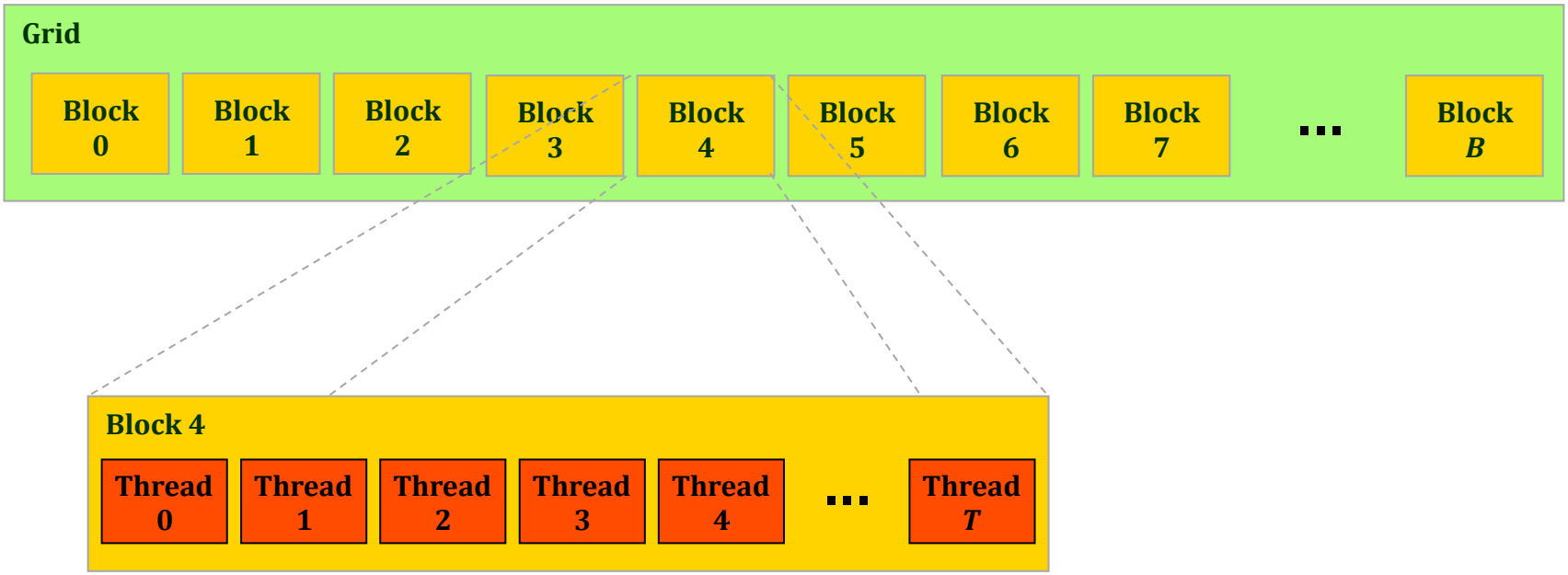
# Thread batching

- To take advantage of the multiple multiprocessors, kernels are executed as a ***grid*** of ***threaded blocks***

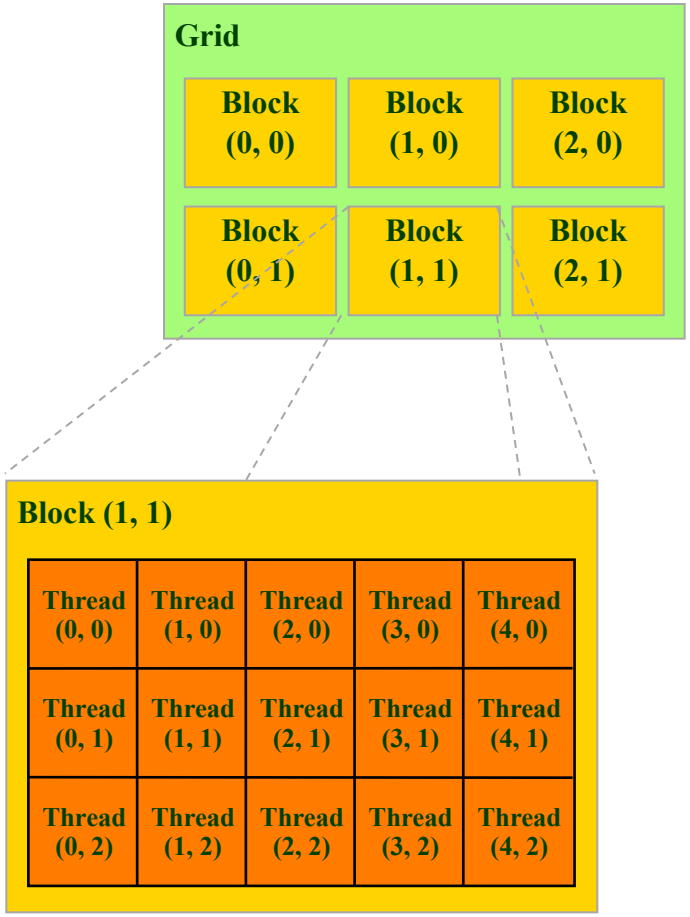
- All threads in a thread block are executed by a single multiprocessor

- The resources of a multiprocessor are divided among the threads in a block (registers, shared memory, etc.)
  - this has several important implications that will be discussed later

# Thread batching: 1D example



# Thread batching: 2D example



# Thread batching (cont.)

- At runtime, a thread can determine the block that it belongs to, the block dimensions, and the thread index within the block

- These values can be used to compute indices into input and output arrays

# Language and compiler

- CUDA provides a set of extensions to the C programming language
  - new storage quantifiers, kernel invocation syntax, intrinsics, vector types, etc.
- CUDA source code saved in `.cu` files
  - host and device code and coexist in the same file
  - storage qualifiers determine type of code
- Compiled to object files using `nvcc` compiler
  - object files contain executable host and device code
- Can be linked with object files generated by other C/C++ compilers

# SAXPY

- SAXPY (Scalar Alpha X Plus Y) is a common linear algebra operation. It is a combination of scalar multiplication and vector addition:

$$\boldsymbol{y = \alpha \cdot x + y}$$

  - $x$ and $y$ are vectors, $\alpha$ is a scalar
  - $x$ and $y$ can be arbitrarily large

# SAXPY: CPU version

- Here is SAXPY in vanilla C:

```
void saxpy_cpu(float *vecY, float *vecX, float alpha, int n)
{
     int i;

     for (i = 0; i < n; i++)
          vecY[i] = alpha * vecX[i] + vecY[i];
}
```

  - the CPU processes vector components sequentially using a for loop
  - note that `vecY` is an in-out parameter here

# SAXPY: CUDA version

- CUDA kernel function implementing SAXPY

```
__global__ void saxpy_gpu(float *vecY, float *vecX, float alpha ,int n)
{
   int i;

   i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i<n)
      vecY[i] = alpha * vecX[i] + vecY[i];
}
```

- The **`__global__`** qualifier identifies this function as a kernel that executes on the device

# SAXPY: CUDA version (cont.)

```
__global__ void saxpy_gpu(float *vecY, float *vecX, float alpha ,int n)
{
   int i;

   i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i<n)
      vecY[i] = alpha * vecX[i] + vecY[i];
}
```

- **`blockIdx`**, **`blockDim`** and **`threadIdx`** are built-in variables that uniquely identify a thread's position in the execution environment
  - they are used to compute an offset into the data array

# SAXPY: CUDA version (cont.)

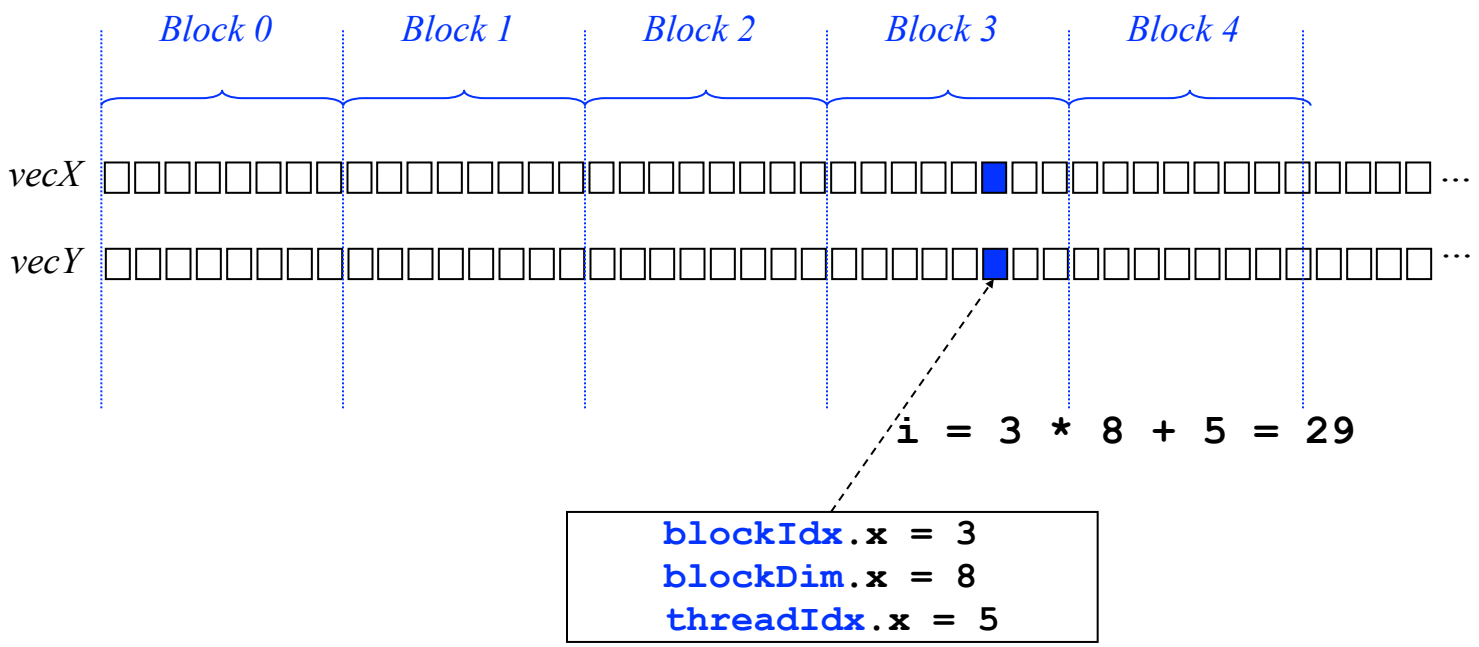
```
__global__ void saxpy_gpu(float *vecY, float *vecX, float alpha ,int n)
{
   int i;

   i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i<n)
      vecY[i] = alpha * vecX[i] + vecY[i];
}
```

- The host specifies the number of blocks and block size during kernel invocation:

```
saxpy_gpu<<<numBlocks, blockSize>>>(y_d, x_d, alpha, n);
```

# Computing the index

```
i = blockIdx.x * blockDim.x + threadIdx.x;
if (i<n)
   vecY[i] = alpha * vecX[i] + vecY[i];
```



# Key differences

```
void saxpy_cpu(float *vecY, float *vecX, float alpha, int n)
{
     int i;

     for (i = 0; i < n; i++)
          vecY[i] = alpha * vecX[i] + vecY[i];
}
```

```
__global__ void saxpy_gpu(float *vecY, float *vecX, float alpha ,int n)
{
   int i;

   i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i<n)
      vecY[i] = alpha * vecX[i] + vecY[i];
}
```

- No need to explicitly loop over array elements – each element is processed in a separate thread
- The element index is computed based on block index, block width and thread index within the block

# Key differences

```
__global__ void saxpy_gpu(float *vecY, float *vecX, float alpha ,int n)
{
   int i;

   i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i<n)
      vecY[i] = alpha * vecX[i] + vecY[i];
}
```

- Could avoid testing whether $i < n$ if we knew $n$ is a multiple of block size (e.g. use padded arrays --- recall MPI_Scatter issues)

```
__global__ void saxpy_gpu(float *vecY, float *vecX, float alpha ,int n)
{
   int i;

   i = blockIdx.x * blockDim.x + threadIdx.x;
   vecY[i] = alpha * vecX[i] + vecY[i];
}
```

# Host code: overview

- The host performs the following operations:
  1. initialize device
  2. allocate and initialize input arrays in host DRAM
  3. allocate memory on device
  4. upload input data to device
  5. execute kernel on device
  6. download results
  7. check results
  8. clean-up

# Host code: initialization

```
#include <cuda.h> /* CUDA runtime API */
#include <cstdio>

int main(int argc, char *argv[])
{
   float *x_host, *y_host;   /* arrays for computation on host*/
   float *x_dev, *y_dev;     /* arrays for computation on device */
   float *y_shadow;          /* host-side copy of device results */

   int n = 1024*1024;
   float alpha = 0.5f;
   int nerror;

   size_t memsize;
   int i, blockSize, nBlocks;

/* here could add some code to check if GPU device is present */

...
```

# Host code: memory allocation

```
...

memsize = n * sizeof(float);

/* allocate arrays on host */

x_host = (float *)malloc(memsize);
y_host = (float *)malloc(memsize);
y_shadow = (float *)malloc(memsize);

/* allocate arrays on device */

cudaMalloc((void **) &x_dev, memsize);
cudaMalloc((void **) &y_dev, memsize);

/* add checks to catch any errors */

...
```

# Host code: upload data

```
...

   /* initialize arrays on host */

for ( i = 0; i < n; i++)
{
   x_host[i] = rand() / (float)RAND_MAX;
   y_host[i] = rand() / (float)RAND_MAX;
}

/* copy arrays to device memory (synchronous) */

cudaMemcpy(x_dev, x_host, memsize, cudaMemcpyHostToDevice);
cudaMemcpy(y_dev, y_host, memsize, cudaMemcpyHostToDevice);

...
```

# Host code: kernel execution

```
...

   /* set up device execution configuration */
blockSize = 512;
nBlocks = n / blockSize + (n % blockSize > 0);

/* execute kernel (asynchronous!) */

saxpy_gpu<<<nBlocks, blockSize>>>(y_dev, x_dev, alpha, n);

/* could add check if this succeeded */

/* execute host version (i.e. baseline reference results) */
saxpy_cpu(y_host, x_host, alpha, n);

...
```

# Host code: download results

```
...

/* retrieve results from device (synchronous) */
cudaMemcpy(y_shadow, y_dev, memsize, cudaMemcpyDeviceToHost);

/* ensure synchronization (cudaMemcpy is synchronous in most cases, but not all) */

cudaDeviceSynchronize();


/* check results */
nerror=0;
for(i=0; i < n; i++)
{
   if(y_shadow[i]!=y_host[i]) nerror=nerror+1;
}
printf("test comparison shows %d errors\n",nerror);

...
```

# Host code: clean-up

```
   ...

   /* free memory on device*/
   cudaFree(x_dev);
   cudaFree(y_dev);

  /* free memory on host */
   free(x_host);
   free(y_host);
   free(y_shadow);

   return 0;
} /* main */
```

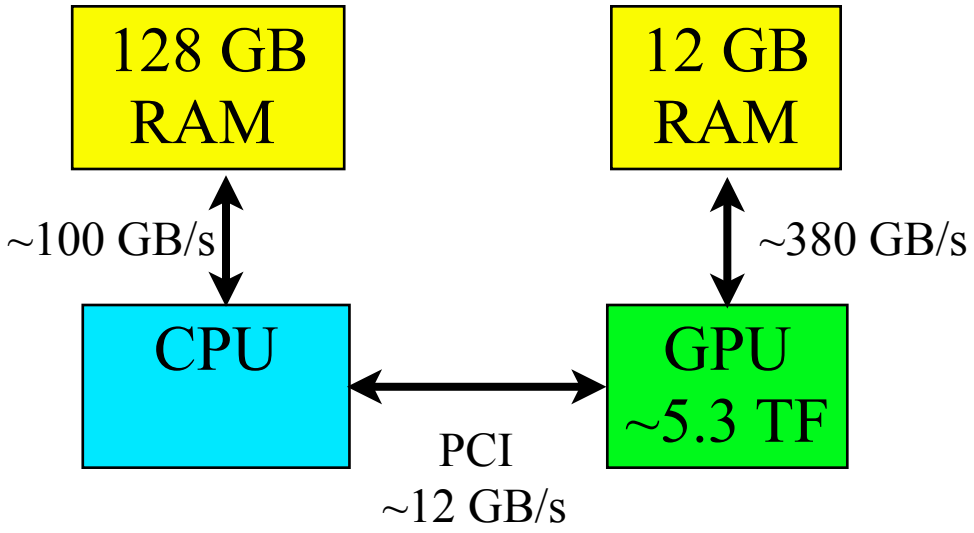
# Checking for errors in CUDA calls

```
 ...
 /* check CUDA API function call for possible error */
if (error = cudaMemcpy(x_dev, x_host, memsize, cudaMemcpyHostToDevice))
  {
    printf ("Error %d\n", error);
    exit (error);
  }

 ...

 saxpy_gpu<<<nBlocks, blockSize>>>(y_dev, x_dev, alpha, n);
 /* make sure kernel has completed*/
 cudaDeviceSynchronize();
 /* check for any error generated by kernel call*/
 if(error = cudaGetLastError())
  {
    printf ("Error detected after kernel %d\n", error);
    exit (error);
  }
```

# Compiling

- nvcc -arch=sm_60 -O2 program.cu -o program.x
- -arch=sm_60 means code is targeted at Compute Capability 6.0 architecture (on graham and cedar)
- -O2 optimizes the CPU portion of the program
- There are no flags to optimize CUDA code
- Various fine tuning switches possible
- Cluster have module installed to provide CUDA environment. See what it does by executing: module show cuda
- add -lcublas to link with CUBLAS libraries

# Be aware of memory bandwidth bottlenecks




- The connection between CPU and GPU has low bandwidth
  - need to minimize data transfers
  - important to use asynchronous transfers if possible (overlap computation and transfer)
  - good idea to test bandwidth (with tool from SDK)

# Using pinned memory

- The transfer between host and device is very slow compared to access to memory within either the CPU or the GPU
- One possibility to speed it up relatively easily is to use pinned memory on the host for memory allocation of array that will be transferred to the GPU
- remember to free this memory correctly

```
cudaMallocHost((void **) &a_host, memsize_input)
...
cudaFree(a_host);
```

# cudaMallocHost

- will allocate CPU memory in a special way which makes it easier for GPU to access
- If both arrays created with cudaMalloc call, can use cudaMemcpyDefault keyword in cudaMemcpy, CUDA will figure out where the data is actually located

```
cudaMallocHost((void **) &x_host, memsize_input);
cudaMalloc((void **) &x_dev, memsize);
cudaMemcpy(x_dev, x_host, memsize, cudaMemcpyDefault);
...
cudaFree(a_host);
```

# Unified Address Space

- presents CPU and GPU memory as a single space
- this does not actually remove the penalty for accessing CPU memory from GPU, and the programmer must be aware of this
- capability evolves, you have to check which generation of card you have to see if these features are supported
- if your Compute Capability is 2.0 or above, the kernel can access data allocated with cudaMallocHost
- sometimes using CPU data in kernel makes the code faster, due to automatic overlapping of computation and memory transfer

# Timing GPU accelerated codes

- Presents specific difficulties because the CPU and GPU can be computing independently in parallel, i.e. asynchronously
- On the cpu can use standard function **gettimeofday(...)** (microsecond precision) and process the result
- If trying to time events on GPU with this function, must ensure synchronization
- This can be done with a call to cudaDeviceSynchronize()
- Memory copies to/from device are synchronized, so can be used for timing.
- Timing GPU kernels on the CPU may be insufficiently accurate

# Using mechanisms on the GPU for timing

- This is highly accurate on the GPU side, and very useful for optimizing kernels

```
...
   cudaEvent_t start, stop;
   float kernel_timer;
...
   cudaEventCreate(&start);
   cudaEventCreate(&stop);
   cudaEventRecord(start, 0);

   saxpy_gpu<<<nBlocks, blockSize>>>(y_dev, x_dev, alpha, n);

   cudaEventRecord(stop, 0);
   cudaEventSynchronize( stop );
   cudaEventElapsedTime( &kernel_timer, start, stop );

   printf("Test Kernel took %f ms\n",kernel_timer);
   cudaEventDestroy(start);
   cudaEventDestroy(stop);
```

# Difficulties with timing properly

- There is overhead to “spinning up” a GPU.
- The very first function call which involves the device may be quite slow as it incorporates the initialization of the GPU.
- The first memory transfer to device is likely to be slower than subsequent ones. The first kernel execution will likely be slower. This will be especially significant if you are trying to time events of short duration.
- A good strategy is to have a “warmup” run: execute your kernel a few times and only then start timing
- To get really good timing, running your kernels repeatedly and obtaining average runtime is essential

# Profiling CUDA - nvprof

- Fortunately, there is a good profiler that comes with CUDA
- It will work on any compiled CUDA executable
- nvcc test.cu -o test.x
  nvprof ./test.x
- It will only time functions involving the GPU, so if you want to compare GPU with CPU performance, you must time the CPU functions yourself

# OpenACC

- New standard for parallel computing developed by compiler makers. See: http://www.openacc-standard.org/
- Specified in late 2011, released in 2012
- OpenACC works somewhat like OpenMP
- Goal is to provide simple directives to the compiler which enable it to accelerate the application on the GPU
- The tool is aimed at developers aiming to quickly speed up their code without extensive recoding in CUDA
- As tool is very new and this course focuses on CUDA, only a brief demo of OpenACC follows
- GCC 5.1 and above now has experimental OpenACC support

# SAXPY with OpenACC

```
...
#include <openacc.h>

void saxpy_openacc(float *restrict vecY, float *vecX, float alpha, int n)
{
     int i;
#pragma acc kernels
     for (i = 0; i < n; i++)
          vecY[i] = alpha * vecX[i] + vecY[i];
}

...
   /* execute openacc accelerated function on GPU */
   saxpy_openacc(y_shadow, x_host, alpha, n);
...
```

- OpenACC automatically builds a kernel function that will run on GPU
- Memory transfers between device and host handled by OpenACC and need not be explicit

# Compiling SAXPY with OpenACC

```
$ module load nvhpc/20.7
$ nvc -acc -ta=tesla  -Minfo=accel saxpy_openacc.c -o saxpy_openacc.x
saxpy_openacc:
     31, Loop is parallelizable
         Generating Tesla code
         31, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     31, Generating implicit copy(vecY[:n]) [if not already present]
         Generating implicit copyin(vecX[:n]) [if not already present]


$ nvprof ./saxpy_openacc.x
==22106== NVPROF is profiling process 22106, command: ./saxpy_openacc.x
test comparison shows 0 errors
==22106== Profiling application: ./saxpy_openacc.x
==22106== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   66.88%  22.696ms        16  1.4185ms  1.4121ms  1.4215ms  [CUDA memcpy HtoD]
                   30.22%  10.256ms         9  1.1396ms  3.1360us  1.2823ms  [CUDA memcpy DtoH]
                    2.89%  981.98us         1  981.98us  981.98us  981.98us
saxpy_openacc_31_gpu
      API calls:   85.86%  289.29ms         1  289.29ms  289.29ms  289.29ms
cuDevicePrimaryCtxRetain
                    9.33%  31.429ms         1  31.429ms  31.429ms  31.429ms  cuMemHostAlloc
…
```

# Is OpenACC always this easy?

- No, the loop we accelerated was particularly easy for the compiler to interpret. It was very simple, and each iteration was completely independent of the others
- If the accelerate directive is placed before a more complicated loop, the compiler refuse to accelerate the region, complaining of errors
- More specific compiler directives must hence be provided for more complicated functions
- Memory transfers must be handled explicitly if we don't want to transfer memory to/from device every time kernel is called
- For complex problems OpenACC grows as complex as CUDA, but it might get better in the future