

Lecture 5

INTRODUCTION TO CUDA

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 *General Purpose computation* on your *Graphics Processing Unit*...

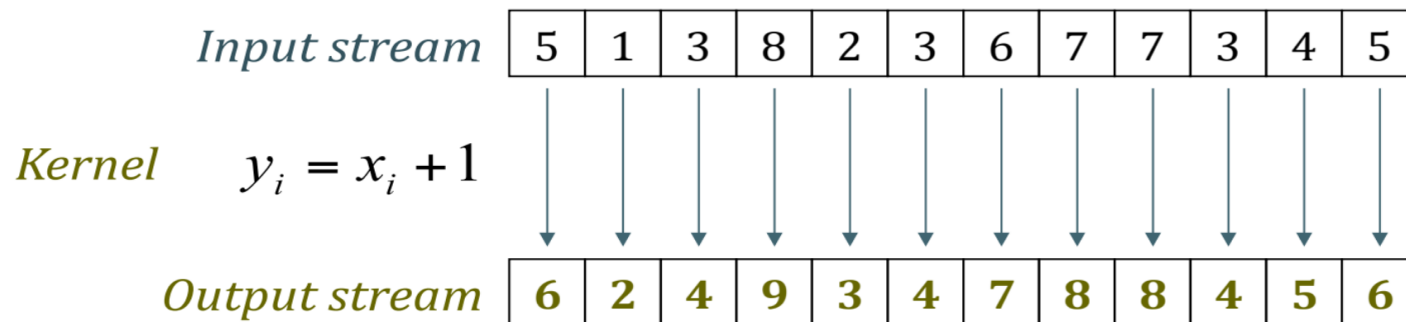


CUDA programming model

- 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

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

Language and compiler

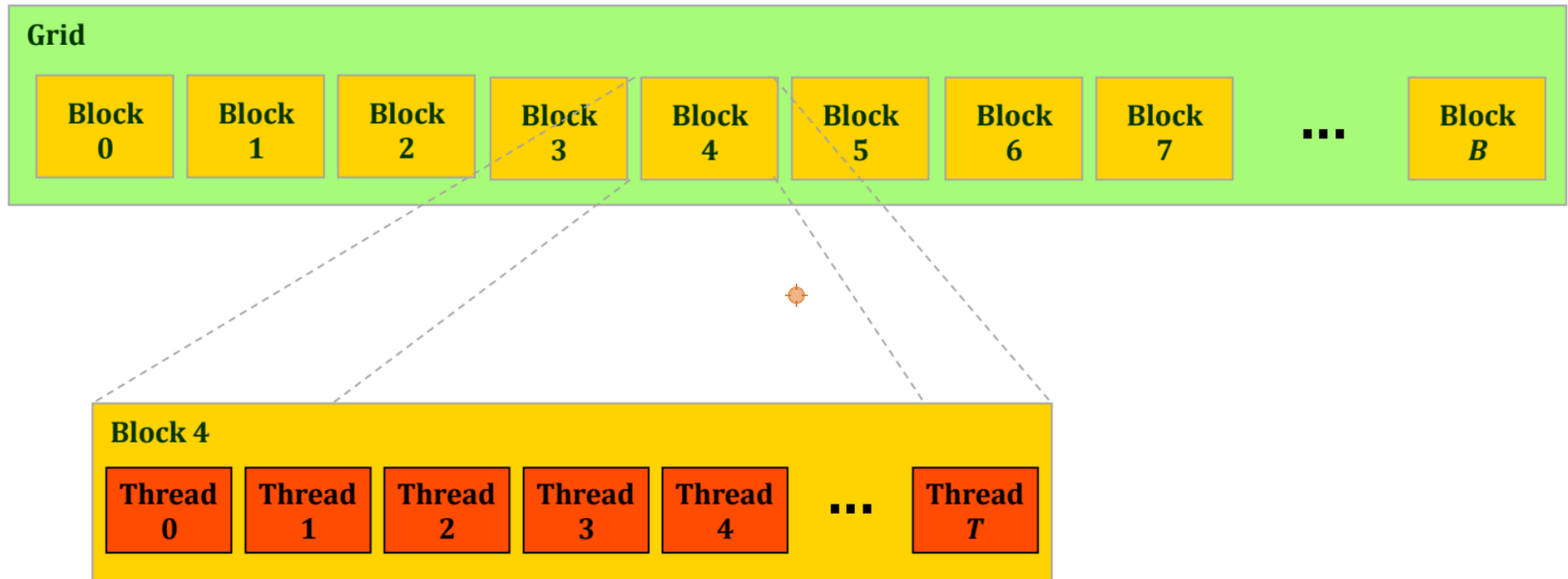
- CUDA provides a set of extensions to the C programming language
 - new storage quantifiers, kernel invocation syntax, intrinsics, vector types, etc.
- CUDA source code saved in `.cu` files
 - host and device code coexist in the same file
 - storage qualifiers determine type of code
- Compiled to object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

Thread batching

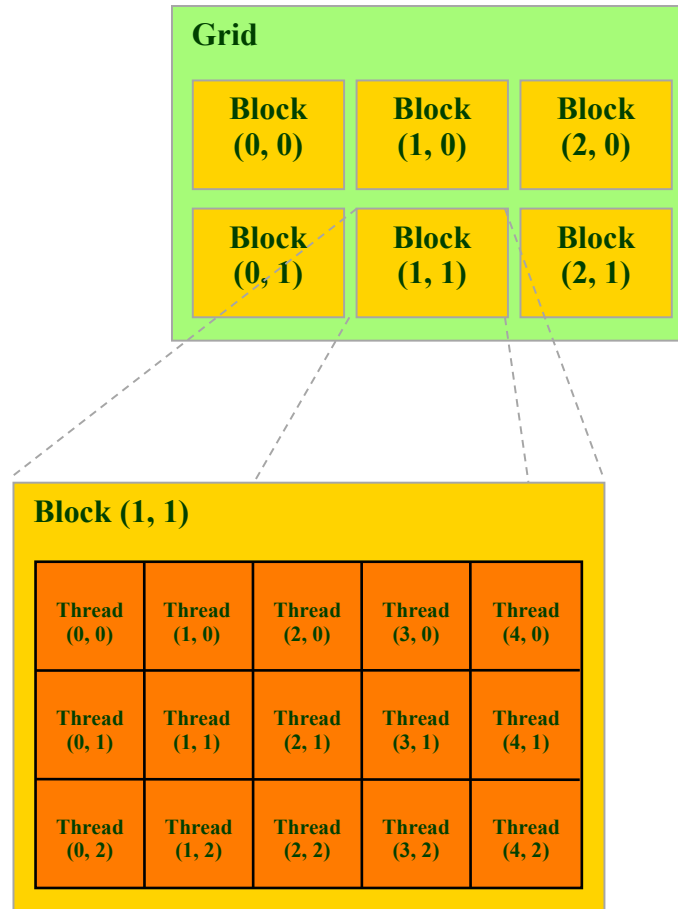
- To take advantage of the multiple multiprocessors, kernels are executed as a *grid of threaded blocks*
- All threads in a thread block are executed by a single multiprocessor
- The resources of a multiprocessor are divided among the threads in a block (registers, shared memory, etc.)
 - this has several 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

SAXPY

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

$$y = \alpha \cdot x + y$$

- x and y are vectors, α 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.)

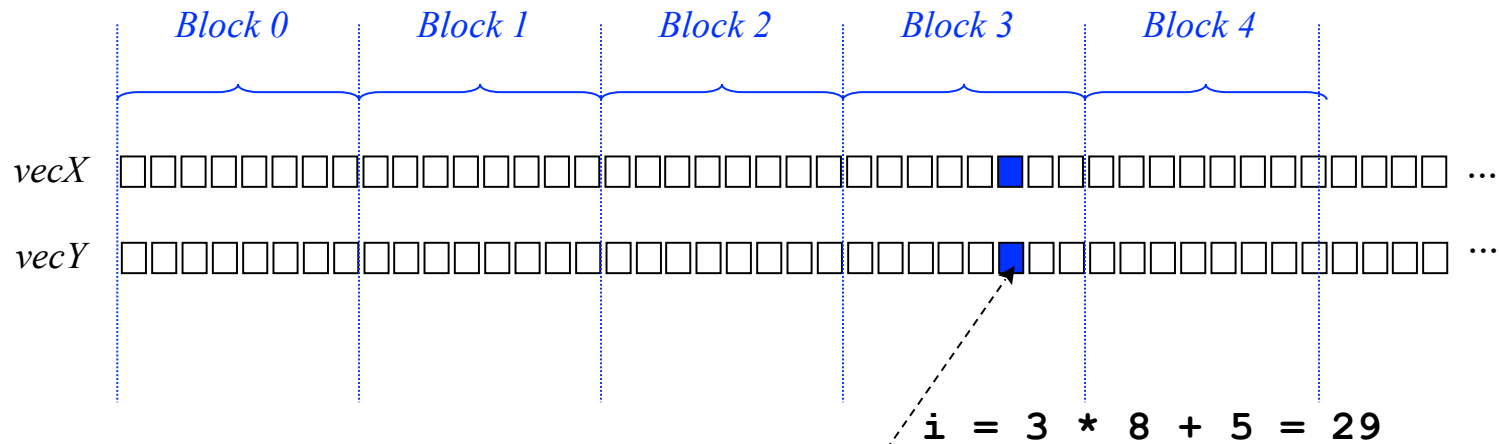
```
__global__ void saxpy_gpu(float *vecY, float *vecX, float alpha, int n)
{
    int i;
    i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        vecY[i] = alpha * vecX[i] + vecY[i];
}
```

- 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];
```



```
blockIdx.x = 3  
blockDim.x = 8  
threadIdx.x = 5
```

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)

```
__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);
}
```



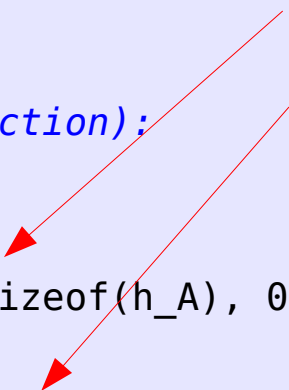

- Dynamic global device arrays

```
// Host code:  
  
// Host array allocation (usually cudaHostAlloc() is a better way):  
size_t size = N * sizeof(float);  
float* h_A = (float*) malloc (size);  
  
// Device array allocation:  
float* d_A;  
cudaMalloc(&d_A, size);  
  
// Host to device copying:  
cudaMemcpy (d_A, h_A, size, cudaMemcpyHostToDevice);  
  
// Device to host copying:  
cudaMemcpy (h_A, d_A, size, cudaMemcpyDeviceToHost);
```

- Static global device arrays

```
// Device code (outside of any function):  
__device__ float d_A[10][50];  
  
// Host code (outside of any function):  
float h_A[10][50];  
  
// Host to device copying:  
cudaMemcpyToSymbol (d_A, &h_A, sizeof(h_A), 0, cudaMemcpyHostToDevice);  
  
// Device to host copying:  
cudaMemcpyFromSymbol (&h_A, d_A, sizeof(h_A), 0, cudaMemcpyDeviceToHost);
```

*Replaces two function calls:
cudaGetSymbolAddress();
cudaMemcpy();*



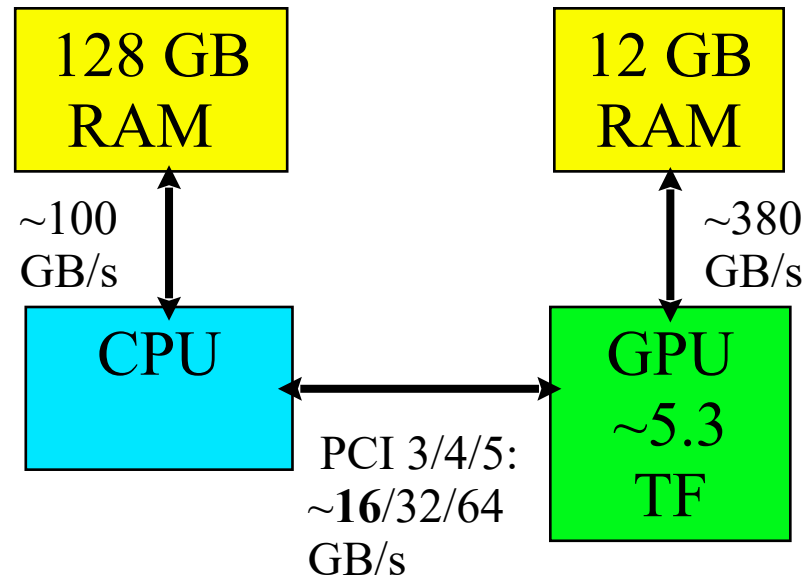
- Device functions

```
// Device code  
  
// Function:  
__device__ double my_function (double x)  
{  
double f;  
...  
return f;  
}  
  
// Kernel:  
__global__ void my_kernel ();  
{  
double f1, x1;  
f1 = my_function (x1);  
}
```

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 were created with cudaMalloc call, one 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 (of course, the same or better performance can be achieved by explicit asynchronous programming).

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 individual 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 (works only up to P100; not for V100 and newer).
- It will work on any compiled CUDA executable

```
$ nvcc test.cu -o test.x
```

```
$ nvprof ./test.x
```

- By default it will only time functions involving the GPU.
 - To time CPU-only codes, add flag “`--cpu-profiling on`” to the `nvprof` command.

Exercise - convert SAXPY code to use `cudaMallocHost`

- Time both the original and converted code
- Estimate what the performance gain is