Distributed and Parallel Computing

Lecture 1

Alan P. Sexton

University of Birmingham

Spring 2018

Task vs Data parallelism

Two types of parallelism

- Task based: do different operations in parallel
  - e.g. Multiply and Add
  - e.g. Gnu Make: supports running sub-tasks in parallel
  - Suitable for standard multi-core CPUs or networks of computers
- Data based: same operations in lock step on different data in parallel
  - e.g. image ops on every pixel of an image simultaneously
  - e.g. dynamical system simulations
  - Suitable for GPUs

Latency vs Throughput

- Latency oriented processors
  - Get each result back with minimum delay
  - Need 100,000 results?
    - Get first back as soon as possible, even if it slows down getting all back
    - Get first of remaining back as soon as possible ...
    - e.g. an operation takes 4 cycles: 10 operations take $4 + 4 + \cdots + 4 = 40$ cycles
- Throughput oriented processors
  - Get all results back with minimum delay
  - Need 100,000 results?
    - Don't care how long it takes to get the first result back, or the second, ...
    - ...so long as the total time to get all back is as small as possible
    - e.g. Implement the operation with deep pipelining on simple ALUs: 10 cycles for the first operation and to fill the pipeline, but 1 cycle for each following operation: $10 + 1 + 1 + \cdots + 1 = 19$ cycles

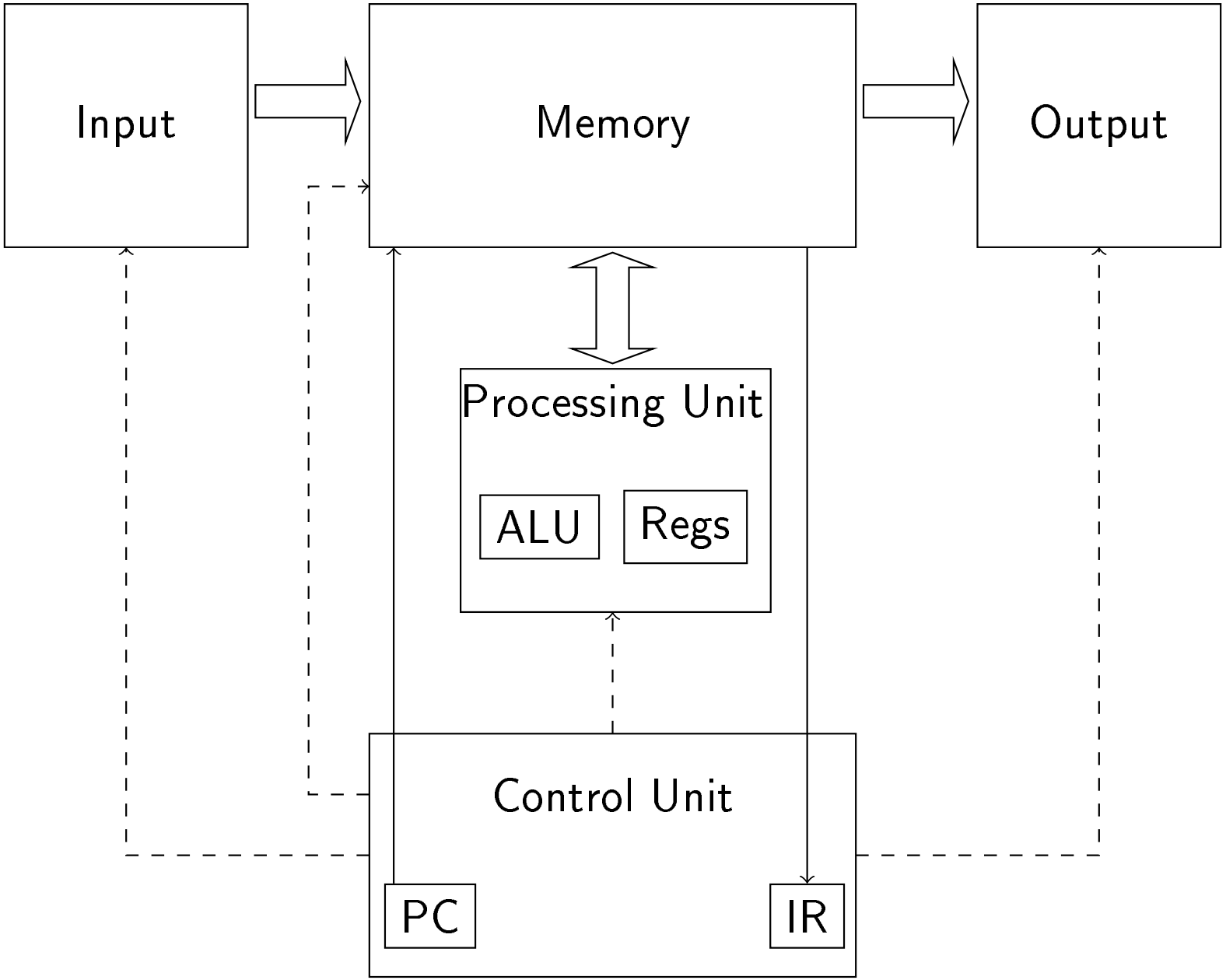
Latency Oriented Processors — standard CPUs

- Large caches to speed up memory access
  - Temporal/Spatial locality, Working sets
    - Try to ensure that each operation has the smallest probability possible of having to fetch from slow memory
- Complex control units
  - Short pipelines, Branch prediction, Data forwarding
    - Jump through hoops to make each instruction finish as quickly as possible with minimal pipeline stalls
- Complex, energy expensive ALUs
  - Large complex transistor arrangements
    - Minimize # Clock Cycles per operation to get result as quickly as possible

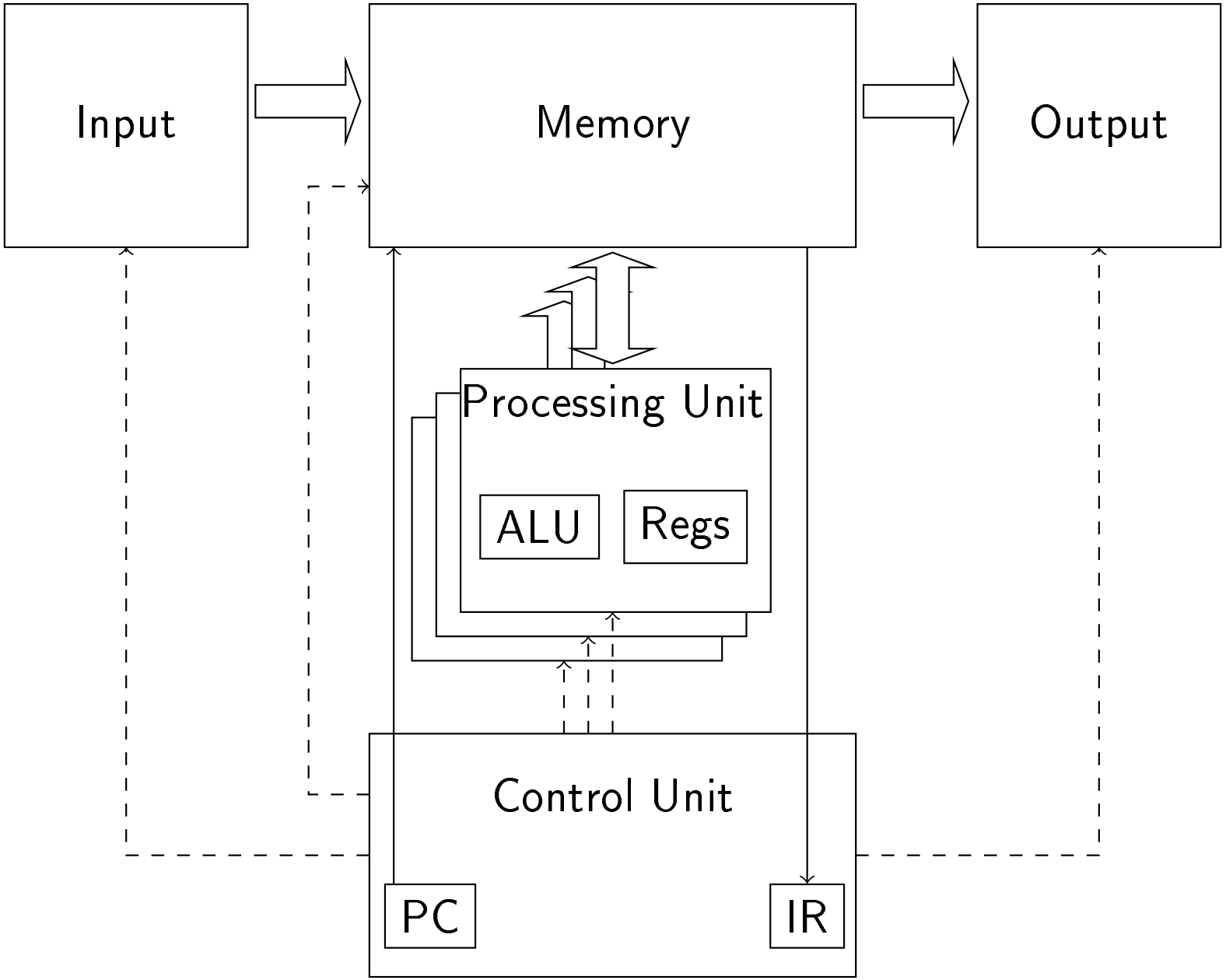
Throughput Oriented Processors — GPUs

- Small caches
  - **NOT** for keeping temporally or spatially located data around
  - For *staging* data
    - Get blocks of data in one go for groups of threads to work on
    - Avoids each thread having to do separate fetches
- Simple control units
  - No branch prediction or data forwarding
  - Control shared across multiple threads operating on different data
- Simple energy efficient ALU
  - Long pipeline
  - Large # cycles per operation but heavily pipelined
    - Long wait for first result (filling the pipeline) but following results come very quickly
  - Requires large numbers of threads to keep the processor occupied

Von Neumann Architecture — standard CPU

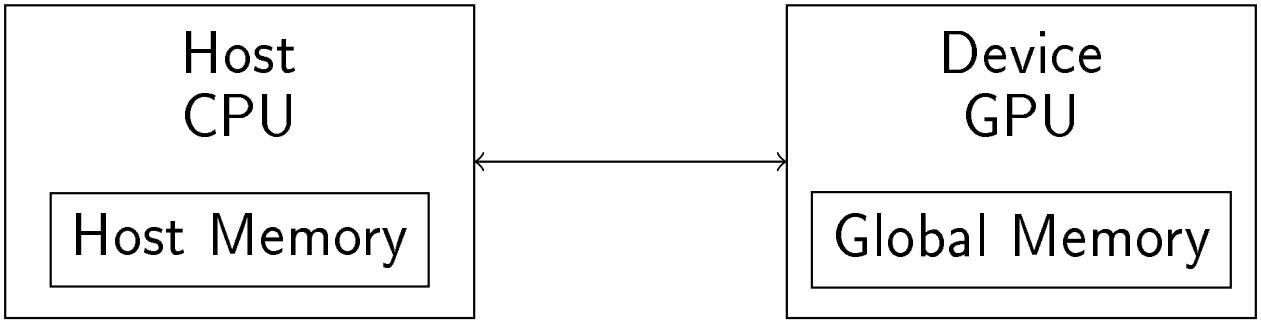


Modified Von Neumann Architecture — GPU



Compiling for CUDA

Conceptually, the host CPU and the GPU are separate devices connected by a communication path:



- Thus we need to generate separate code for each device. The NVidia compiler for CUDA programs is *nvcc*
- nvcc takes a C or C++ program with NVidia extensions, separates out and compiles the GPU Device code itself, and separates out the Host code and passes it to the local host compiler to be compiled.
- The single resulting binary contains both the host and the device binary, which is downloaded to the Device from the Host when the program runs
- While you can compile with `nvcc` just as with `gcc` or `g++`, it is easier to use `nsight`, the NVidia modified, CUDA enabled Eclipse IDE.

VectorAdd

As a first CUDA code example, we will look at adding two vectors:

$$C = A + B$$

In sequential C, the code might look like this:

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    for (i = 0; i < n ; i++)
        h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Declare and set value for N
    // Declare and allocate memory for h_A, h_B and h_C
    // Populate h_A and h_B with data
    vecAdd(h_A, h_B, h_C, N);
    ...
}
```

Programming in CUDA

We have to identify whether a function runs on the host, the device or both, and where it is callable from:

- the host: `__host__ void f(...)`
  - This is the default so can be omitted
  - Callable from the host only
- the device: `__global__ void f(...)`
  - These special functions are called *kernel functions*
  - Callable from the host only, hence this is how the host gets to run code on the GPU
- the device: `__device__ void f(...)`
  - Callable from the device only. Hence they are helper function available to kernel functions and other device functions on the GPU
- both: `__host__ __device__ void f(...)`
  - This generates both a host function and a device (i.e. helper) function so the same code can be run on both. The host can not call the device version or vice versa

GPU computational unit structure

When we call a kernel function, we need to specify how the threads should be organised to execute it. Note:

- Every thread is going to execute the same kernel. For our vector addition example, we will get each thread to add a single element of A to a single element of B to produce a single element of C.
- Different GPU devices can support different numbers of simultaneously executable threads.
- Within a GPU, it would be nice if we could make the thread structure uniform: i.e. if every processing unit had equal access to all the GPU memory, could synchonise equally with any other processing unit and could share all the GPU cache memory. However, we could only do this by sacrificing a huge proportion of potential computational power.

CUDA thread issues

- We don't want the GPU fixed number of threads to dictate the size of the largest vectors we can add
- We don't want to have to change our code to run the program on a different GPU with a different number of threads
- We need to be able to organise our threads to co-locate groups of threads on sets of processing units to take advantage of shared caches and synchronisation facilities.

NVidia GPUs accomplish this by organising threads into a hierarchical structure:

- A *Grid* is a collection of *Blocks*
- A *Block* is a collection of *Threads*
- A *Thread* is the execution of a *kernel* on a single processing unit

CUDA thread organisation

Outside the Grid/Block/Thread hierarchy, there is the concept of a *Warp*

- A *warp* is a set of a number of tightly related threads that must execute fully in lock step with each other.
- Warps are not part of the CUDA specification, but are a feature of all NVidia GPUs, dictated by low level hardware design
- The number of threads in a warp is a feature of a particular GPU, but with current GPUs it is almost always 32
- Warps are the low-level basis of thread scheduling on a GPU. If a thread is scheduled in to execute, all the other threads in that warp are scheduled in too
- As they execute the same instructions in lock step, all threads in a warp will have exactly the same instruction execution timing

Grid/Block/Thread

- A block can have between 1 and the maximum block size number of threads for that GPU device (typically 1024 for our machines) and is the high-level basis for thread scheduling
- Because of the nature of warps, the block size should be a multiple of the warp size. Otherwise blocks will be padded with the remaining threads in the partially used warp and wasted
- Grids can have very large numbers of blocks, many more than can be executed at once

Grid/Block/Thread

- The grid corresponds to the whole problem (vector add of 50,000 elements) divided up into bitesize blocks (e.g. vector add of 1024 elements)
- a GPU that can execute one block at a time (1024 threads) would schedule $\lceil 50,000/1024 \rceil$ blocks, one at a time, hence a minimum of 48 schedulings
- a GPU that can execute three blocks at a time (1024 threads) would schedule $\lceil 50,000/1024 \rceil$ blocks, three at a time, hence a minimum of 17 schedulings

Invoking Kernel Functions

We need to specify the grid/block structure when invoking a kernel function

```
...
int threadsPerBlock = 256;
int blocksPerGrid = 1 + (numElements-1) / threadsPerBlock;
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C,
     numElements);
...
```

Note that the "`<<<blocksPerGrid, threadsPerBlock>>>`" is not standard C or C++, but is handled by the `nvcc` compiler.

Inside Kernel Functions

Each thread needs to know which part of the data to work on. CUDA provides predefined variables for this purpose:

- `blockIdx.x`: the unique identifier of this block in this grid
- `blockDim.x`: the number of threads in a block for this grid
- `threadIdx.x`: the unique identifier of this thread in this block (between 0 and blockDim.x - 1)

```
__global__ void
vectorAdd(const float *A, const float *B, float *C, int n)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < n)
    {
        C[i] = A[i] + B[i];
    }
}
```

Grid and Block Dimensionalities

- Grids and Blocks can be organised as 1-dimensional (suitable for our vector add example), 2-dimensional (suitable for operating on pixels of an image) or 3-dimensional (suitable for operating 3-dimensional spatial simulations)
- Hence the predefined variables, `blockIdx.x`, `blockDim.x`, `threadIdx.x` have ".y" and ".z" variants as well.
- If you are using one dimensional grids and blocks (i.e. you set the grid and block size of the kernel using simple integers), then you can ignore the ".y" and ".z" variants

Device Global Memory

Memory on a GPU is not (currently) shared with the Host machine. Hence the host has to copy data to the device and copy it back when the kernel finishes

Before doing so, the host must allocate global memory on the device and, afterwards, free it again, just like `malloc` and `free` in C

Note that, unlike `malloc`, the return value of `cudaMalloc` is an error number

```
float *h_A = (float *)malloc(size);
float *h_C = (float *)malloc(size);
float *d_A = NULL;
err = cudaMalloc((void **)&d_A, size);
float *d_C = NULL;
err = cudaMalloc((void **)&d_C, size);
...
err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
... // invoke kernel ...
err = cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
...

err = cudaFree(d_A);
err = cudaFree(d_C);
```

CUDA Error Handling

The only way to check that things are working correctly on the GPU is to check the error return values. *ALWAYS* check them *EVERY* time.
The standard code to check error numbers is:

```
if (err != cudaSuccess)
{
    fprintf(stderr,
            "SUITABLE ERROR MESSAGE (error code %s)!\n",
            cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
```

CUDA Error Handling

- Kernel functions don't return error numbers. However after it has finished, you can call `err = cudaGetLastError();` to get the error number if an error occurred.
- Since kernel functions can run in parallel with host functions, if you call `cudaGetLastError()` before the kernel function finishes, the error may only occur after you requested the error, leading to a confusing error later on.
- If you really want to be sure it finished correctly (either to check for errors or because you are timing it, or you want to extract the final results from the device), call `cudaDeviceSynchronize()`
- Normally you should not unnecessarily call `cudaDeviceSynchronize()`, because allowing a sequence of kernel calls to work without unnecessary extra synchronisation is more efficient.

Timing Host Code with Host Timers

Typically we will want to time both the sequential version of the code that runs on the Host CPU and the parallel version that runs on the GPU.
The general timing approach for host code is as follows:

```
#include <cuda_runtime.h>
#include <helper_cuda.h>
#include <helper_functions.h>

StopWatchInterface *timer = NULL;
sdkCreateTimer(&timer);                    // create a timer
sdkStartTimer(&timer);                      // start the timer

/* The Host code that is to be timed */

sdkStopTimer(&timer);
double h_msecs = sdkGetTimerValue(&timer);
sdkDeleteTimer(&timer);
```

Timing Host+GPU Code with Host Timers

If using host timers to time GPU code, recall that GPU code runs asynchronously with host code, so make the host wait until the GPU has finished before stopping the timer:

```
#include <cuda_runtime.h>
#include <helper_cuda.h>
#include <helper_functions.h>

StopWatchInterface *timer = NULL;
sdkCreateTimer(&timer);                    // create a timer
sdkStartTimer(&timer);                      // start the timer

/* The Host+GPU code that is to be timed */

cudaDeviceSynchronize();

sdkStopTimer(&timer);
double h_msecs = sdkGetTimerValue(&timer);
sdkDeleteTimer(&timer);
```

Timing GPU Code with GPU Timers

The best way to time GPU code, is to insert an **event** into the GPU execution stream before and after the code to time and get the elapsed time from them:

```
cudaEvent_t start, stop;
float d_msecs;

cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord( start, 0 );

/* Call GPU kernel(s) */

cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop );

cudaEventElapsedTime( &d_msecs, start, stop );
cudaEventDestroy( start );
cudaEventDestroy( stop );
```

Time will be in milliseconds with a resolution of approximately 0.5 milliseconds