GPU programming with CUDA

Andrew Emerson,

with contributions from Luca Ferrara and Sergio Orlandi

What are GPUs ?

Graphics Processing Units (GPU) were originally designed for doing graphics but now routinely used in HPC and machine learning applications.

Very different design to conventional CPUs, optimised for *streaming* computations.

Most important vendors for HPC and machine learning are NVIDIA who also developed the CUDA language.

Why use GPUs?

➕ **Advantages**

- Very large number of “cores” means potentially very high performance increase compared to CPU applications;
- Often low cost ($/Flop) and low energy consumption (Watts/Flop)

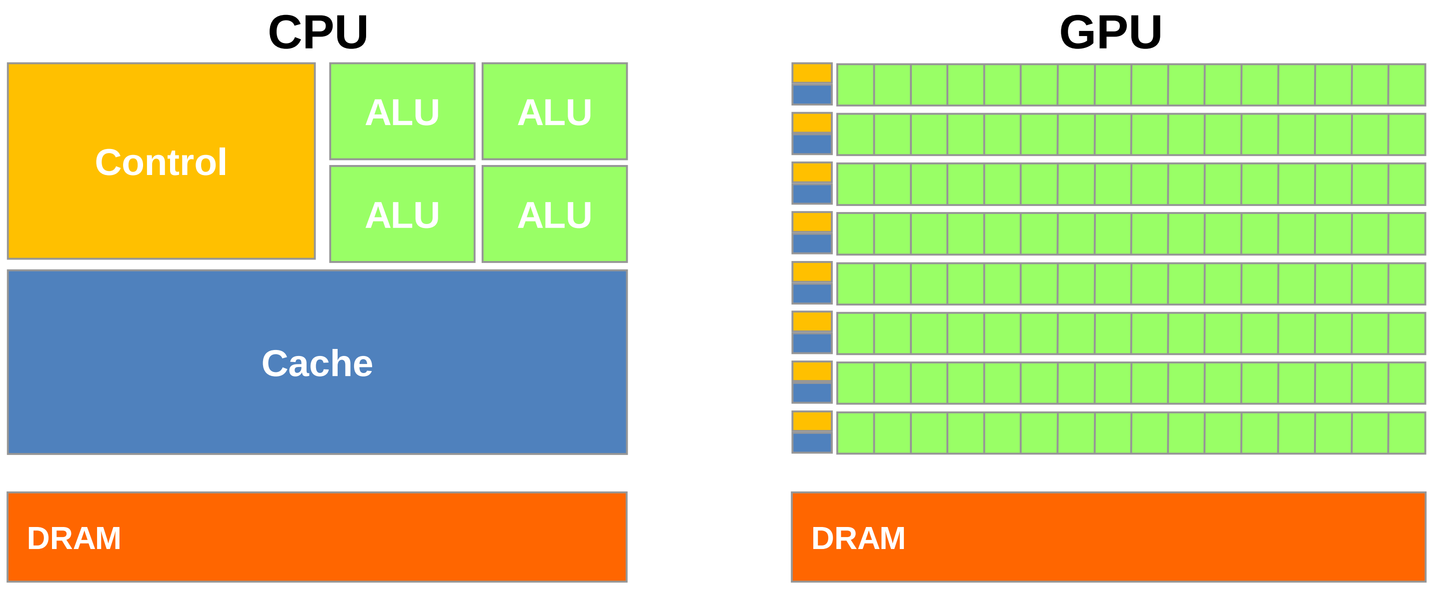
➖ **Disadvantages**

- May need complete rewrite of application;
- Low device memory (max 16/32 Gb)
- Depending on model, possible low transfer speeds.

CPU vs GPU Architectures

GPU hardware is designed so that more transistors are devoted to data processing rather than data caching and flow control

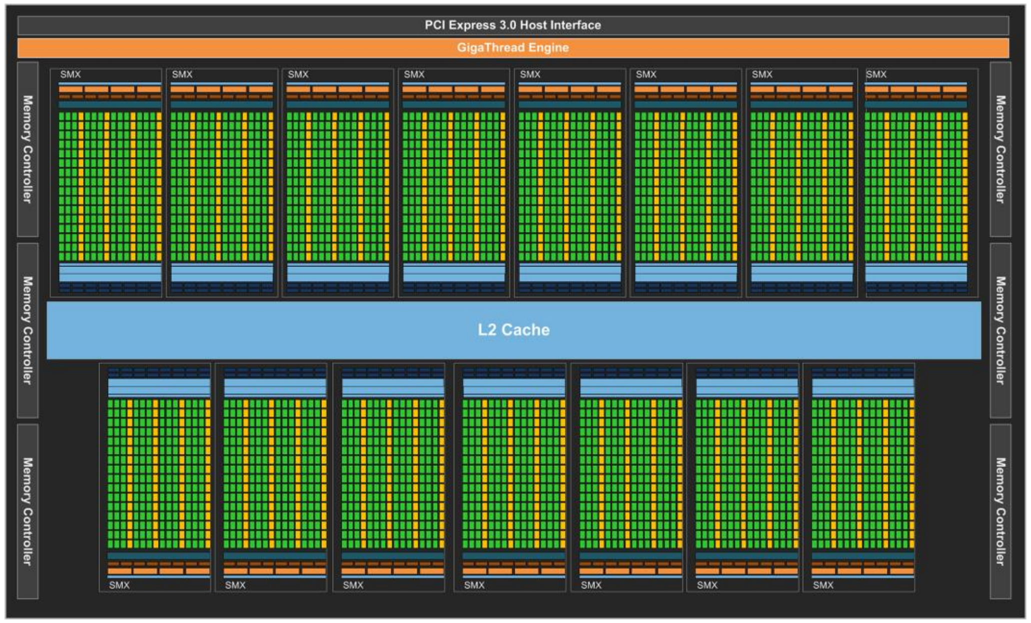
– specialized for problems which can be classified as *intense data-parallel computations* where the same set of operations are executed on different data



10's of threads

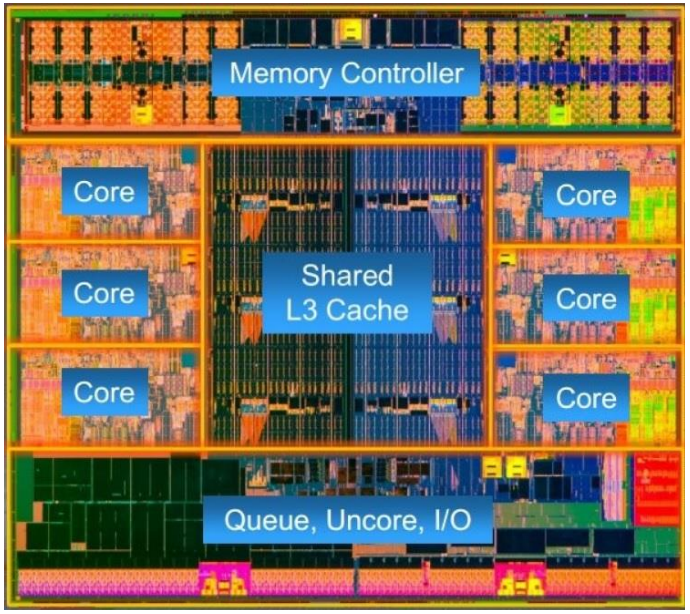
10000's of threads

Processor die comparison



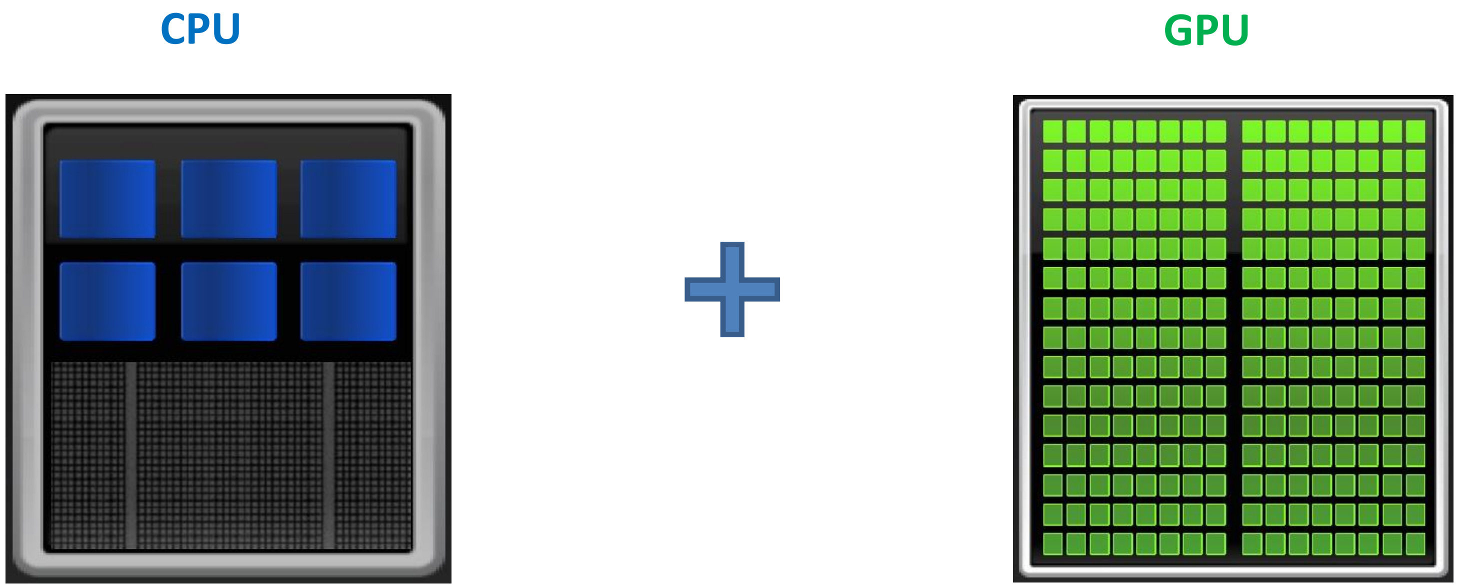
NVIDIA GPU Kepler (2013)

Intel Core i7-4960X



GPGPU Programming Model

- General Purpose GPU Programming relates to use GPU computational power to solve problems other than graphics
- CPU and GPU are **separate devices** with **separate memory** space addresses
- GPU is seen as an auxiliary coprocessor equipped with thousands of cores and a high bandwidth memory
- They should work together for best benefit and performances

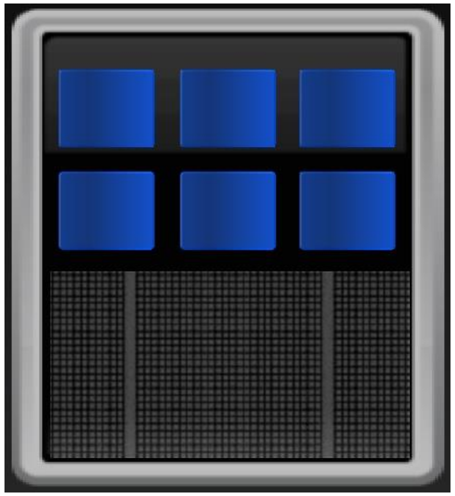


GPGPU Programming Model

- Optimized for low-latency accesses to caches data sets
- Control logic for out-of-order and speculative execution
- Best for serial or event driven tasks

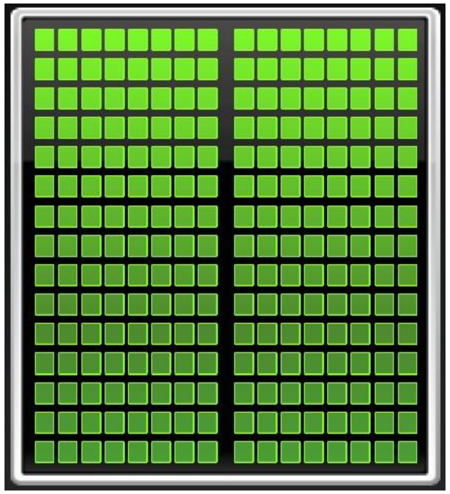
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- Best for data-parallel tasks

CPU



+

GPU



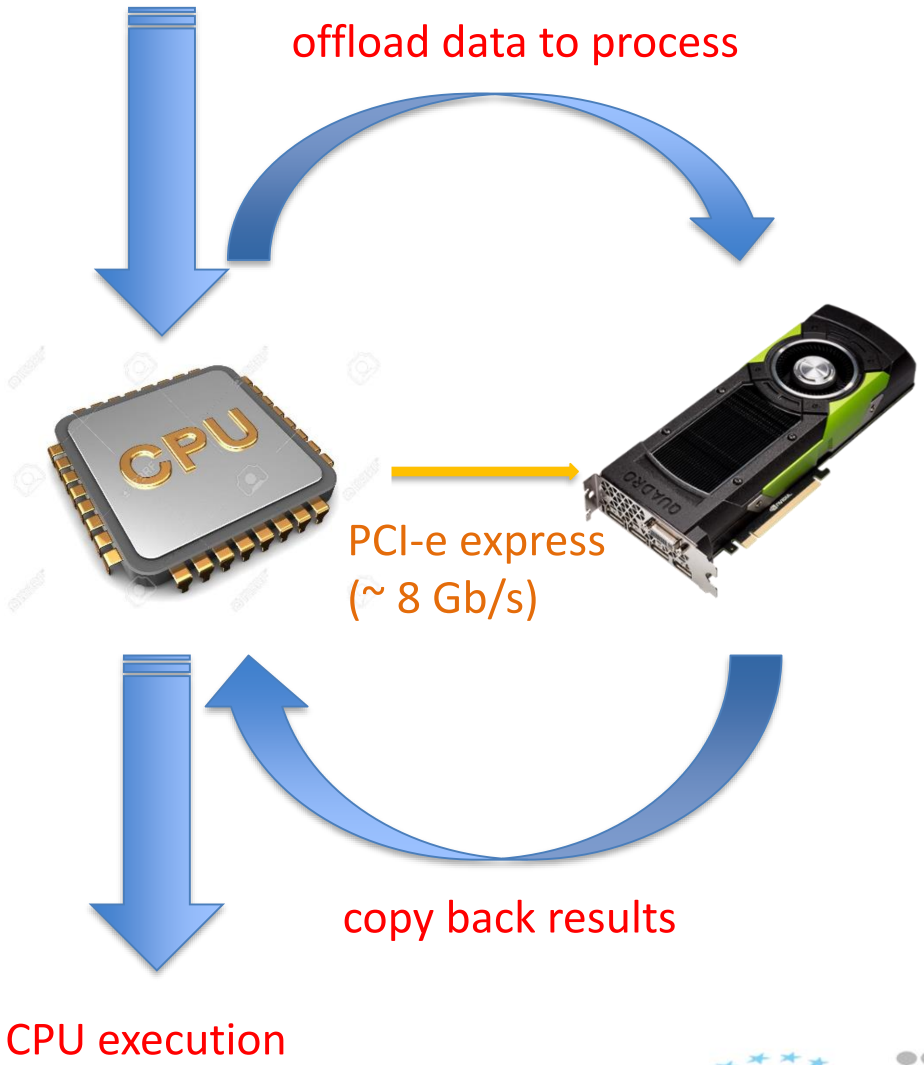
GPGPU programming model

Serial, or processes with low parallelism, remain on the CPU, while highly parallel operations are *offloaded* to the GPU for processing. They are then copied back.

Connection to device is relatively slow so minimise data transfers.

The GPU's threads need to be *saturated* otherwise the result will be slower than CPU.

For max performance, execution on the CPU should continue as much as possible while GPU is being used.



How do I program GPUs?

The situation is changing rapidly but possibilities include:

Declarative languages

- OpenMP
  - v 4.0+ allows offloading of tasks onto GPUs
- OpenAcc
  - High-level model, particularly suited for devices such as GPUs.

Languages

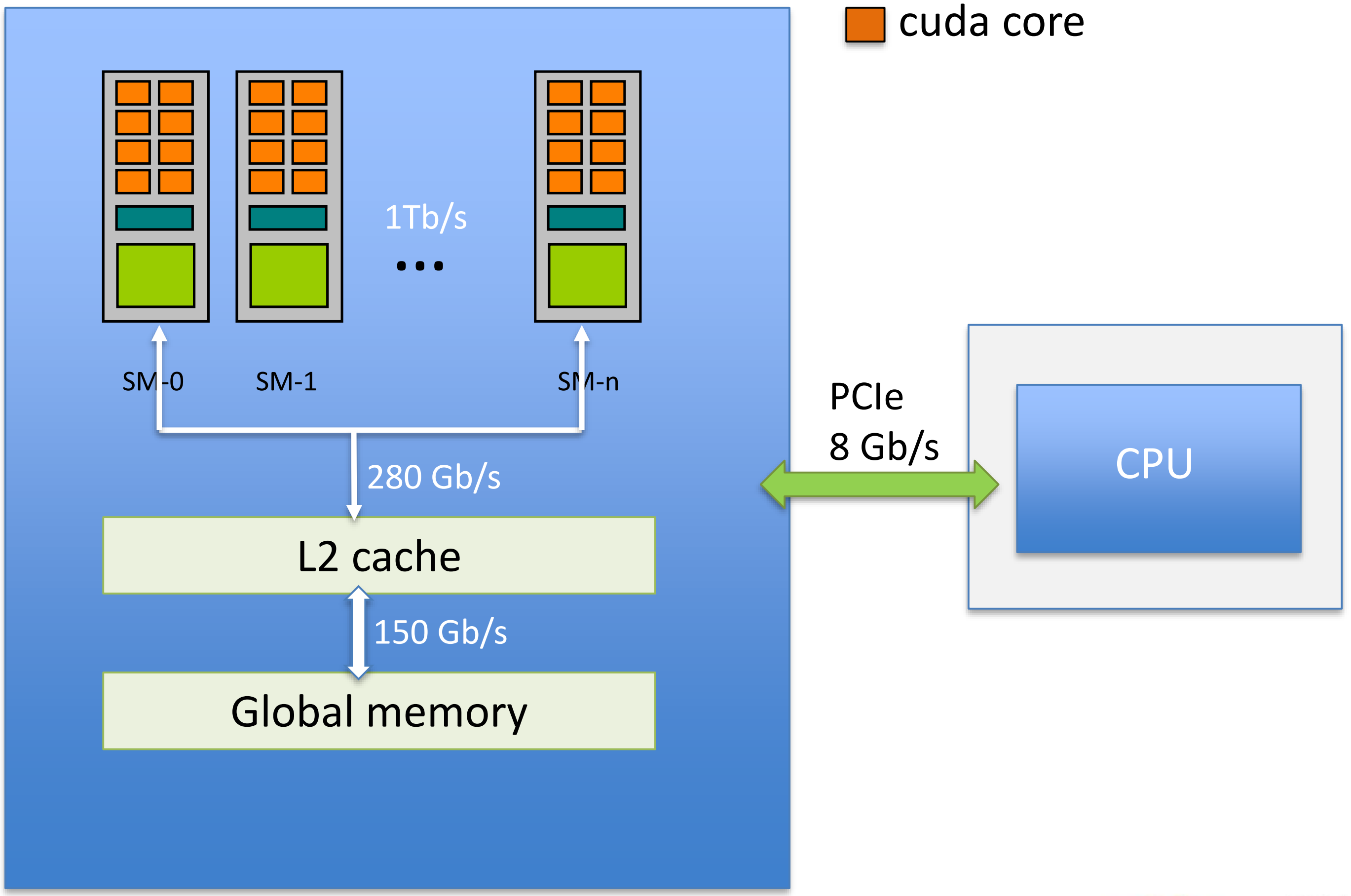
- CUDA
  - Extension to C developed by NVIDIA. With PGI compilers, FORTRAN extension also possible.
- OpenCL
  - General framework for writing programs across heterogenous devices. Often used for non-NVIDIA GPUs and FPGAs.

CUDA: Compute Unified Device Architecture

- Parallel computing platform and API created by NVIDIA for programming on GPUs.
- Originally only for C/C++ but now FORTRAN API is available with PGI compilers.
- First version of the SDK (Software Development Kit) arrived in 2007 (CUDA-1) - in 2019 we now have CUDA -10.
- The SDK includes:
  - Drivers, runtimes and API
  - Compiler wrappers for compiling cuda code (nvcc)
  - Libraries (cuBLAS, cuFFT, cuSolver, etc)
  - Debuggers (cuda-gdb, cuda-memcheck), profilers (nvprof, nView), etc.
- Freely downloadable from NVIDIA, but cuda-enabled GPUs are available only from NVIDIA.
- Fastest solution when using NVIDIA devices.

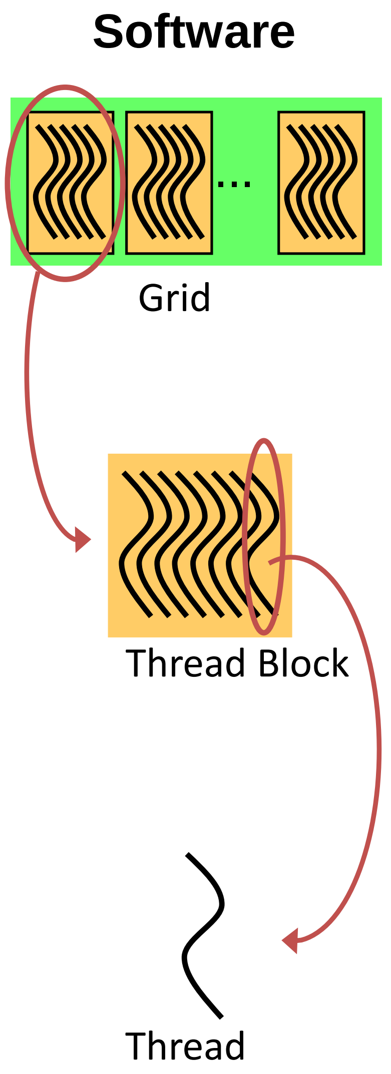
NVIDIA GPU Achritecture

GPU consists of many Streaming Multiprocessors (SM), each containing GPU cores and shared memory.



Kepler architecture

the GPU Execution Model



Grid: a collection of thread blocks

- Thread blocks do not synchronize with each other.
- Communication between blocks is expensive.

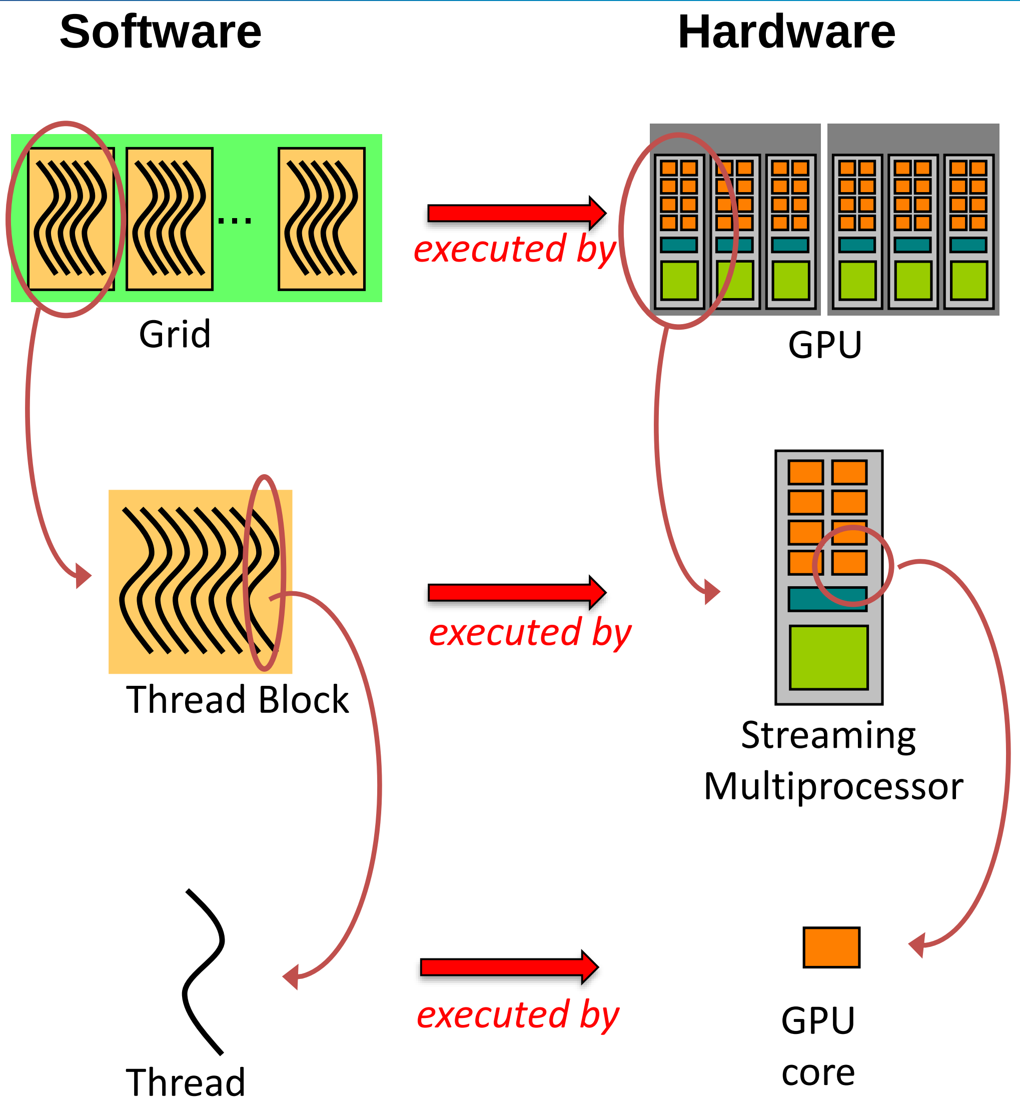
Thread Block: a group of threads

- Threads within a block can cooperate (light-weight synchronization).

Thread: a sequential execution unit

- All threads execute same sequential program.
- Threads execute in parallel.
- A “warp” is a set of 32 threads which execute the same instruction.

the GPU Execution Model



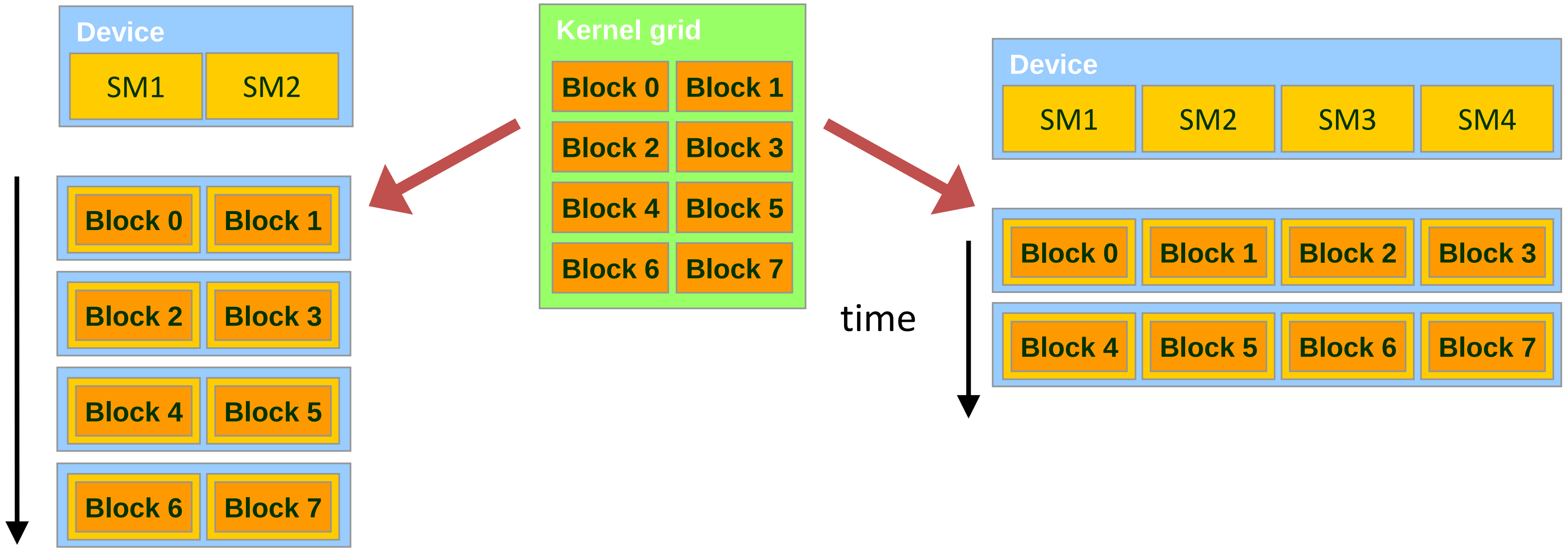
When a CUDA kernel is invoked:

- thread blocks are assigned to SMs in a round robin mode.
- the thread block remains on the SM until all its threads have finished (remember, no communication between thread blocks).
- threads of each thread block are partitioned into warps of consecutive threads
- each thread in a warp executes the same instruction simultaneously, with each thread being managed by one GPU core.

Trasparent Scalability

The GPU runtime system can execute blocks in any order relative to each other

This flexibility enables to execute the same application code on hardware with different numbers of SMs.

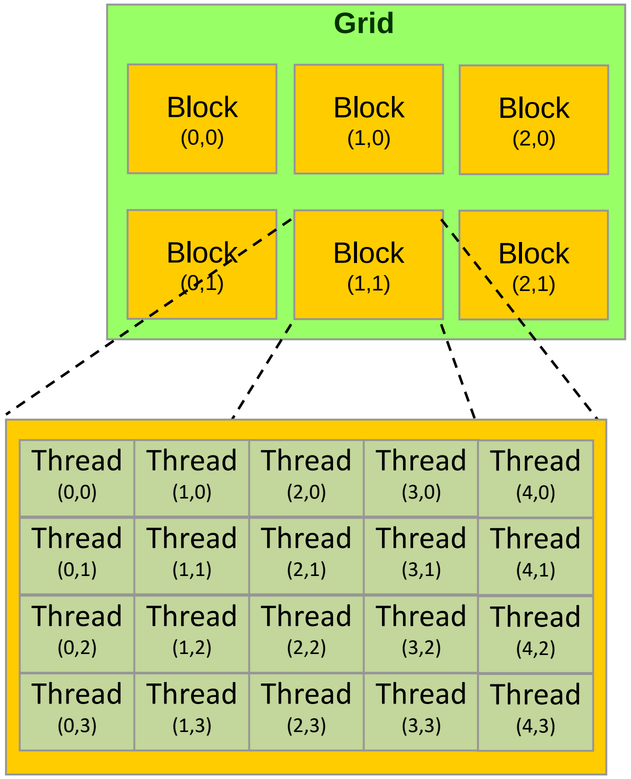


CUDA thread indexing

In CUDA it is possible to launch kernels with 1D, 2D or 3D arrays of thread blocks within the Grid and threads within the block.

At runtime CUDA will define variables showing how the blocks are arranged in the Grid and the threads within each block, together with block and thread *indices*.

Grid and Block Dimensions	
gridDim.x, gridDim.y, gridDim.z	Number of blocks in the x,y and z dimensions
blockDim.x, blockDim.y, blockDim.z	Number of threads in the x, y and z dimensions
Block and Thread index	
blockIdx.x, blockIdx.y, blockIdx.z	Block's index in x, y and z dimensions
ThreadIdx.x, ThreadIdx.y, ThreadIdx.z	Thread's index in the x, y and z directions (of the Block)



CUDA thread indexing

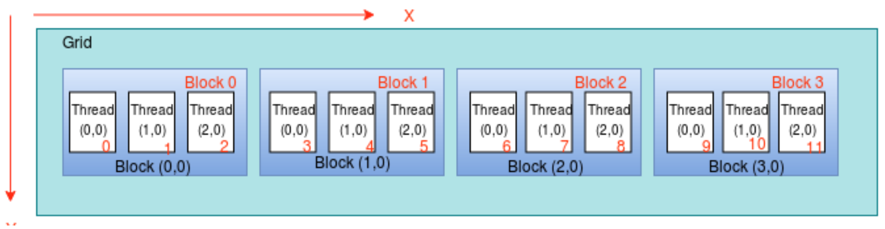
For the programmer usually important to know the *unique* or *global* index of a thread.

This can be calculated for the different block and thread topologies using the CUDA variables.

Grid and Block Dimensions	
gridDim.x, gridDim.y, gridDim.z	Number of blocks in the x,y and z dimensions
blockDim.x, blockDim.y, blockDim.z	Number of threads in the x, y and z dimensions
Block and Thread index	
blockIdx.x, blockIdx.y, blockIdx.z	Block's index in x, y and z dimensions
ThreadIdx.x, ThreadIdx.y, ThreadIdx.z	Thread's index in the x, y and z directions (of the Block)

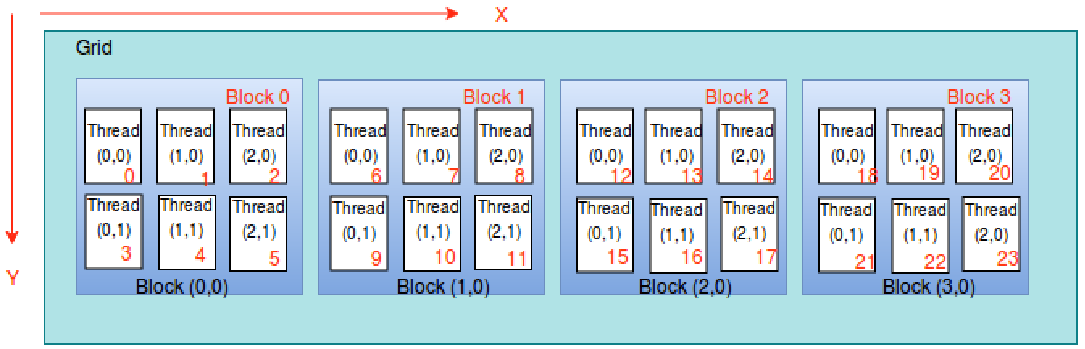
EXAMPLES

1D grid of 1D blocks



```
threadId=(blockIdx.x*blockDim.x)+threadIdx.x
```

1D grid of 2D blocks



```
threadId=(blockIdx.x*blockDim.x*blockDim.y) +
(threadIdx.y*blockDim.x) + threadIdx.x
```

CUDA and NVIDIA GPUs

How many threads and blocks can I use?
Depends on the *compute capability* of the device which describes the GPU features available.

Code name	Product name	Compute capability	SMM units	Max threads/ block	Max thread blocks/s m	#cores (FP32)
Kepler (GK210)	Tesla K40	3.7	15	1024	16	2496
Maxwell	Tesla M40	5.2	24	1024	32	3072
Pascal	Tesla P100	6.0	56	1024	32	3584
Volta	Tesla V100	7.0	80	1024	32	5120

But often makes sense to set threads/block =1024 and make the number of blocks = problem_size/1024

Writing CUDA programs

To write a CUDA C program you do something similar to the following:

1. Declare and allocate host and device memory
2. Initialize host data
3. Copy data from host to device
4. Execute one or more kernels
5. Transfer results from the device to the host

Clearly, you need to know what parts of the code should be written as kernels.

First CUDA program

```
#define N 512
#define THREADS_PER_BLOCK 512
int main( void ) {
 int *a, *b, *c; // host copies of a, b, c
 int *dev_a, *dev_b, *dev_c; // device copies of a, b, c
 int size = N * sizeof( int ); // we need space for N integers
 int i;

// allocate device copies of a, b, c
 cudaMalloc( &dev_a, size );
 cudaMalloc( &dev_b, size );
 cudaMalloc( &dev_c, size );

 // allocate host arrays
 a = (int*)malloc( size );
 b = (int*)malloc( size );
 c = (int*)malloc( size );

//Initialise a,b arrays
for (i=0;i<N;i++)
   a[i]=b[i]=1;

 // copy inputs to device
 cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice);
 cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice);

 // launch add() kernel on GPU, passing parameters
 add<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>( dev_a, dev_b, dev_c);

 // copy device result back to host copy of c
 cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);
```

```
__global__ void add( int *a, int *b, int *c) {

int index = threadIdx.x + blockIdx.x * blockDim.x;

if (index <N)
  c[index] = a[index] + b[index];

}
```

Memory Allocation and Copying

Code	Meaning

`int size=N*sizeof(int);`
`int *dev_a;`
`cudaMalloc(&dev_a, size );`
	• allocate on the device size bytes (in this case N ints).
	• note the double pointer

`cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice);`
`cudaMemcpy( a, dev_a, size, cudaMemcpyDeviceToHost);`
	• copy array a from host to array dev_a on device
	• copy array dev_a from device to array a on host

`cudaFree(dev_a);`	• Free memory on device associated with array dev_a.

The cudaMemcpy calls will wait until previous CUDA kernels have finished and will block until the data has been transferred.

CUDA kernels

```
__global__ void add( int *a, int *b, int *c) {

 int index = threadIdx.x + blockIdx.x *
blockDim.x; // global thread id

 if (index <N)
   c[index] = a[index] + b[index];

}
```

- Kernel functions indicated in the code by __global__ (called by the host) or __device__ (called by another function on the device).
- Must be void - cannot return values
- Remember that every CUDA thread executes the code in the function. May need to use **if** statements to make sure unallocated memory is not accessed.

Launching CUDA kernels from host

```
add<<< THREAD_BLOCKS, THREADS_PER_BLOCK >>>(dev_a, dev_b, dev_c);
```

- Just like a call to a standard C function (with no return value) but need to specify the kernel configuration, i.e the number of thread blocks and threads in a block (block size), in the <<<>>>.
- The kernel function needs the __global__ identifier.
- The call is **non-blocking**, i.e. will return almost immediately. In this way you can overlap GPU and CPU execution.
- To make sure kernel launches are synchronised you can call cudaMemcpy() or invoke explicitly cudaDeviceSynchronize();

For a general problem size N very often the following formula is used to give the minimum number of thread blocks required:

thread blocks=(N +blocksize/1)/blocksize

Compiling and running a CUDA program

```
module load cuda
nvcc -arch=sm_37 -o add add.cu
# log onto a node with a GPU
./add
```

- Compile CUDA programs with nvcc, a wrapper of another C compiler, called the host compiler (often gcc, but could be something else).
- nvcc processes and compiles the CUDA sections, while non-CUDA code is forwarded to the host compiler.
- CUDA source is first compiled to *PTX* assembly language, before being converted by the driver to binary for the GPU.
- On some systems important to specify the GPU architecture (--arch/--gpu-architecture) otherwise CUDA code won't be compiled. The option will determine the level of CUDA available to your code and the number of threads and thread blocks available.

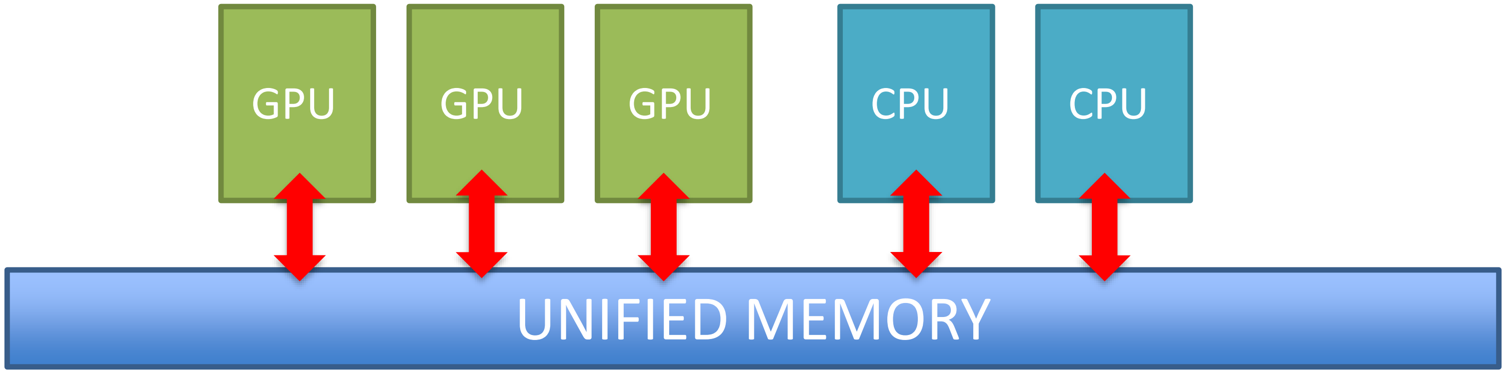
Other CUDA command line programs

- nvidia-smi
  - Shows which GPUs are available and gives information about them
  - Can be used in scrolling mode when running CUDA programs
- nvprof
  - Quick profiler, useful for showing memory transfers between host and device.
  - More sophisticated profiling can be done with nvvp.
- cuda-memcheck
  - Ideal for spotting memory leaks in the CUDA program. Will considerably slow execution.
- cuda-gdb
  - CUDA debugger

Further CUDA

Unified Memory

- Allows allocated data to be read or written by either CPUs or GPUs, i.e. no need to explicit copy data.
- Complete implementation available from Pascal P100, while Kepler K80 has a limited version.
- Requires some care or could be slower than explicit copying.
- Remember this is really *"virtual"* memory- data still has to be copied over the PCIe link.



```
cudaMallocManaged(&x,N*sizeof(float));
```

further CUDA

Shared Memory

- Fast, on-chip memory with very low latency
- Allocated per thread so all threads in the block have access to the shared memory.
- Threads can access data in shared memory loaded from global memory by other threads in the same threads block.
- Useful when performing parallel reduction (with thread synchronization).
- Can be defined at compile or at run time - if runtime, added as a third parameter to the kernel configuration and use extern keyword.

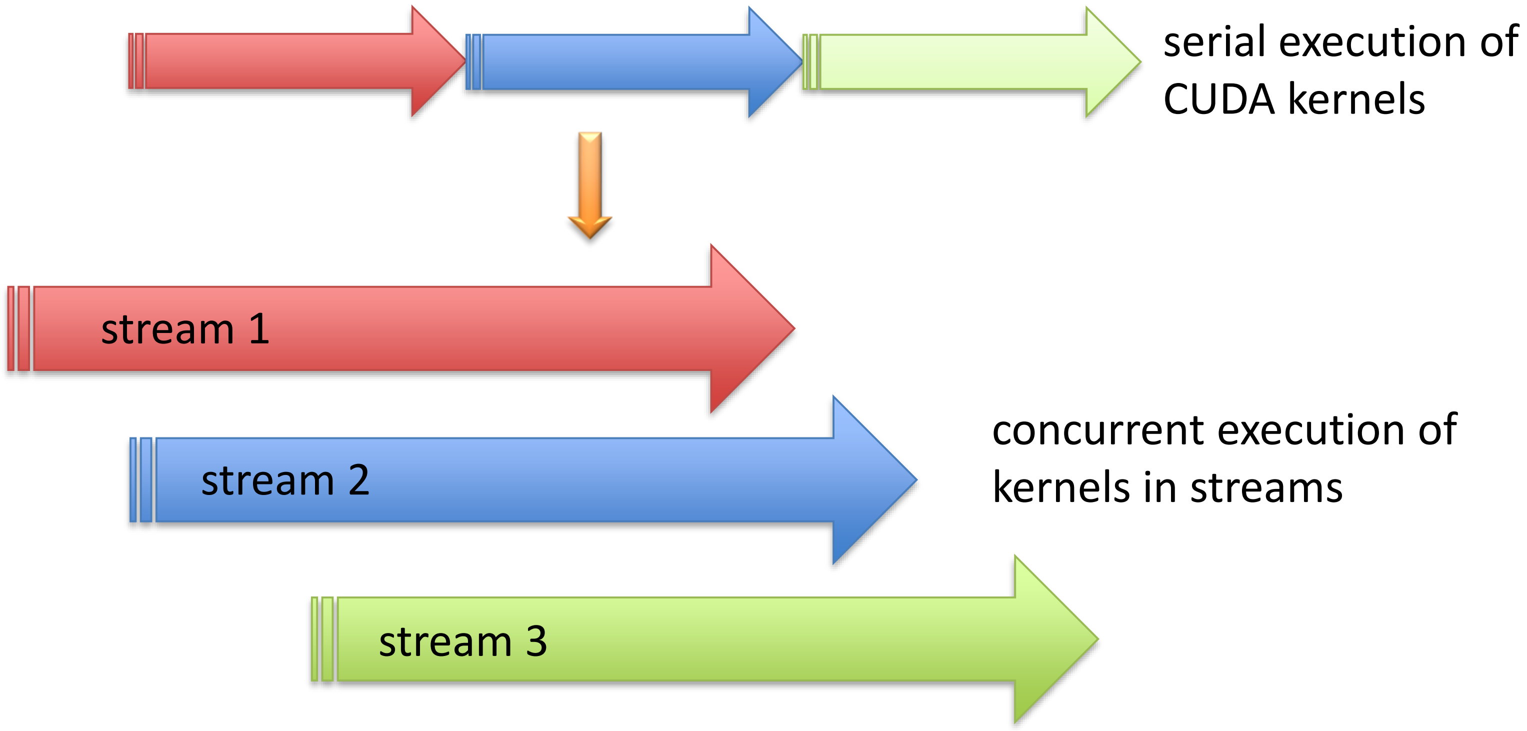
```
__kernel__
void kernel(double *x, int n) {
extern __shared__ double sdata[];
    int tid=threadIdx.x;
    sdata[tid]=0.0
    …
}

kernel<<<1,n,n*sizeof(int)>>>(x,n)
;
```

thread synchronization

- With __syncthreads() possible to synchronize threads within a thread block
- often used with shared memory to avoid race conditions

```
__global__ void reverse(int *d,int n)
{
    extern __shared__ int s[];
    int t=threadIdx.x;
    int tr = n-t-1;
    s[t] =d[t];
    __syncthreads();
    d[t] = s[tr];
}
```

further cuda - CUDA streams


serial execution of CUDA kernels
stream 1
stream 2
stream 3
concurrent execution of kernels in streams

- CUDA (at least capability 2.x) allows kernels to be launched in *streams*.
- The commands in each stream are executed in order but the streams are asynchronous with respect to each other.
- This allows much greater concurrency in the CUDA code.

CUDA FORTRAN

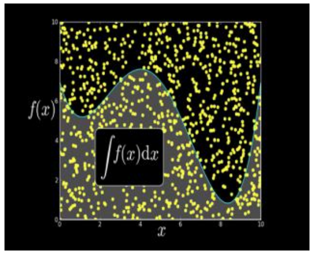
- Recent introduction due to collaboration with NVIDIA and PGI. Best current implementation of CUDA FORTRAN is with PGI compilers.
- Same concepts of threads and thread blocks as CUDA C (e.g threadidx%x, blockid%y, etc..). Kernels are subroutines.
- Variables allocated on the GPU identified by device attribute, i.e. no explicit copy.
- Particularly useful feature is automatic kernel generation by CUF kernels (cf with OpenAcc).

```
program testramp
use cublas
use ramp
integer, parameter :: N = 20000
real, device :: x(N)
twopi = atan(1.0)*8
call buildramp<<<(N-1)/512+1,512>>>(x,N)
!$cuf kernel do
do i = 1, N
x(i) = 2.0 * x(i) * x(i)
end do
print *,"float(N) = ",sasum(N,x,1)
end program
```

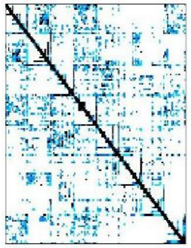
```
module ramp
real, constant :: twopi
contains
attributes(global) &
subroutine buildramp(x, n)
integer ::I
i=threadidx%x
….
end subroutine
end
```

Some GPU-accelerated Libraries

NVIDIA cuBLAS



NVIDIA cuRAND


NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

Vector Signal Image Processing

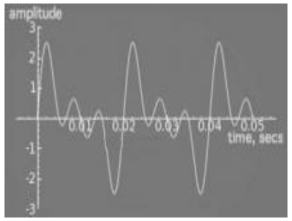
CULA | tools

GPU Accelerated Linear Algebra

MAGMA

Matrix Algebra on GPU and Multicore

open source initiative



NVIDIA cuFFT

ROGUE WAVE SOFTWARE

IMSL Library

ArrayFire Matrix Computations

CUSP

Sparse Linear Algebra

open source initiative

Thrust

C++ STL Features for CUDA

open source initiative