The Abdus Salam
International Centre for Theoretical Physics

United Nations
Educational, Scientific and
Cultural Organization

IAEA
International Atomic Energy Agency

1967-22

Advanced School in High Performance and GRID Computing

3 - 14 November 2008

Introduction to GPU programming in the nvidia CUDA environment

BASHEER Ershaad Ahamed

Jawaharlal Nehru Centre for Advanced Scientific Research
Centre for Computational Materials Science
Jakkur P.O., Bangalore 560064
Karnataka
INDIA

ICTP

Advanced School for High Performance
and GRID Computing

Introduction to GPU programming in the NVIDIA CUDA environment

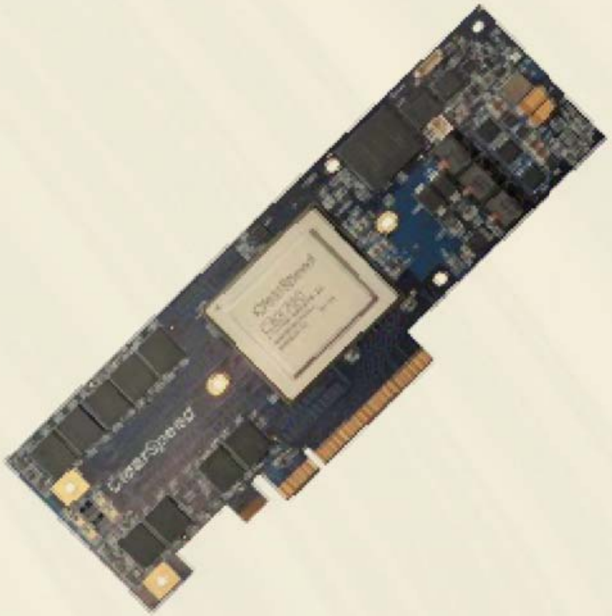
Ershaad Ahamed

**Jawaharlal Nehru Centre
for Advanced Scientific Research
Bangalore, India**

ershaad@jncasr.ac.in

Hardware Acceleration

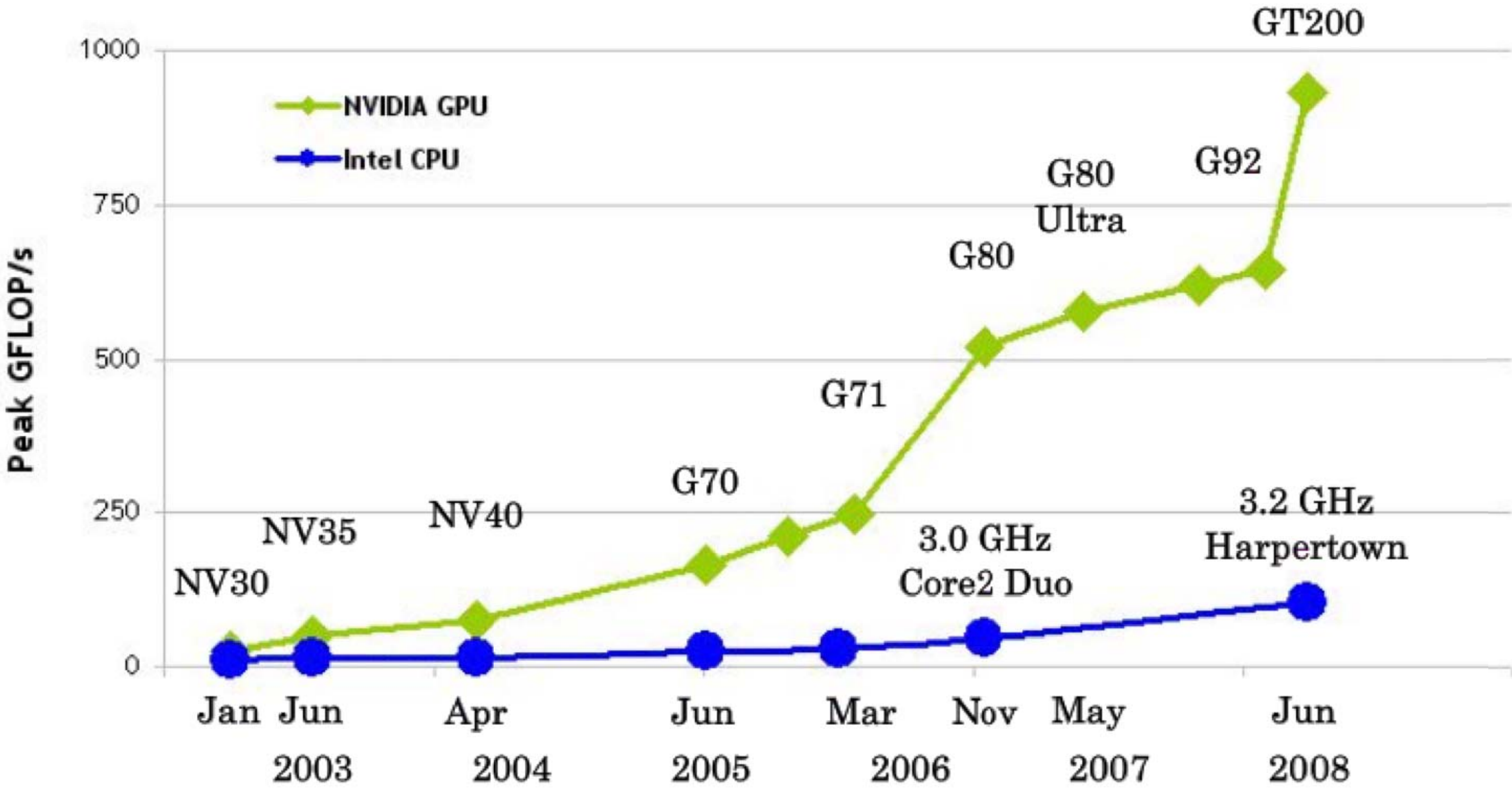
- Use of hardware to perform some function faster than is possible in software running on the general purpose CPU



GPU

- GPU – Graphics Processing Unit
- Demand for higher quality real-time graphics with affordable hardware
- Designed specifically for accelerating graphics processing
- GPU has become highly parallel with high memory bandwidth

GeForce GTX 280 has a 240 core GPU with 141 GB/sec of GPU memory bandwidth source: NVIDIA



GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	

GPU

- GPU's workload is highly parallel floating point operations
- Programmability has been added to the processing pipelines of modern GPUs
- The floating point pipelines can be used to perform non-graphics related floating point operations.

GPU vs. CPU

How does the GPU achieve such high performance figures?

- CPU approach
  - Designed to maximize serial performance
  - Achieves this using several methods
  - Code doesn't need to be modified to take advantage of better chip design (except multicore)

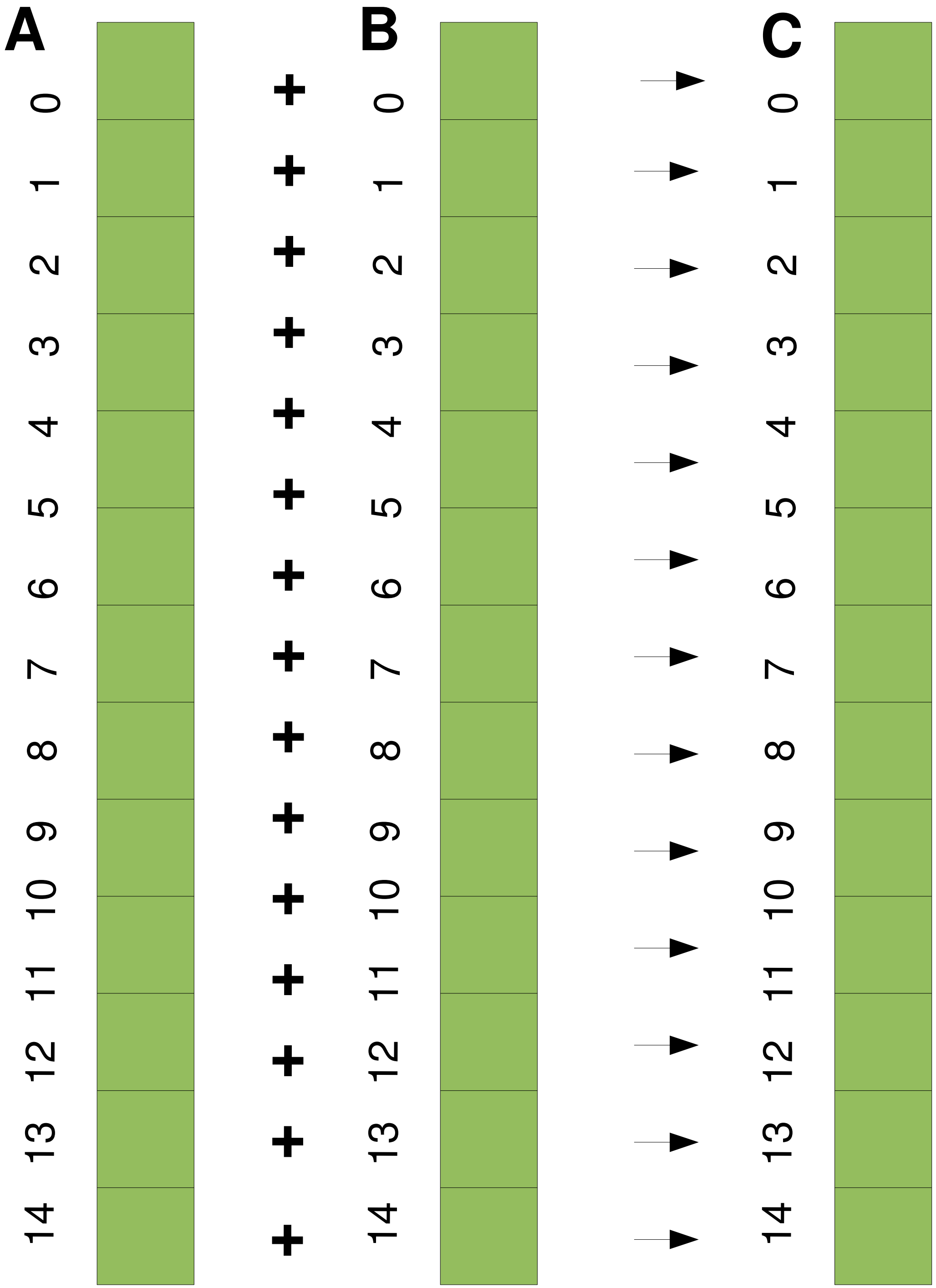
GPU vs. CPU

- GPU approach
  - Prefers to exploit parallelism over serial performance
  - Do away with the circuitry needed for serial performance (data caching, branch prediction...)
  - Devote more die space for “ALUs” (cores)
  - Programs need to be written specifically to take advantage of the hardware
  - CUDA (Compute Unified Device Architecture) is the programming model developed by NVIDIA for programming their GPUs

source: NVIDIA

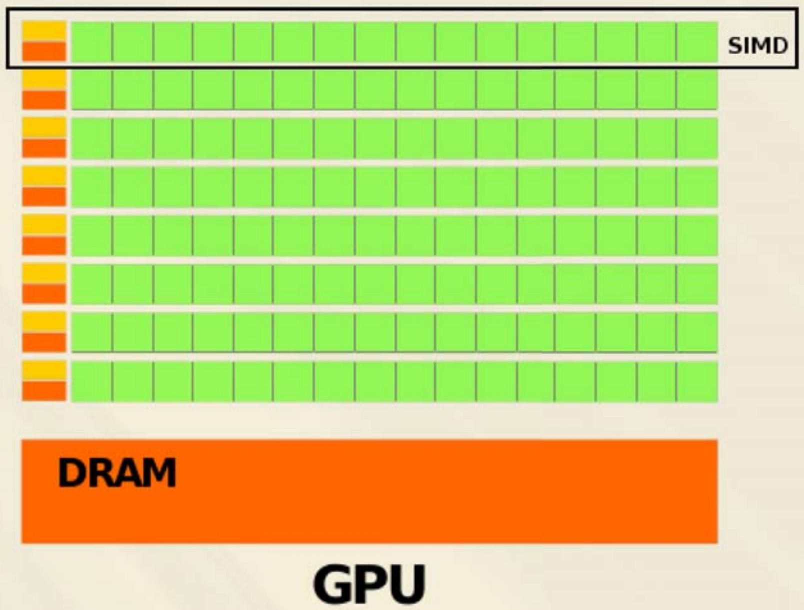
Data parallelism

- Perform the same operation on several data elements in parallel
- Example: find the sum of corresponding elements of two arrays


A
B
C
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14

Data parallelism in the GPU

- Each pair of data elements in processed separately by a processing unit called a Scalar Processor (SP)
- SPs are grouped into SIMD units called Streaming Multiprocessors (SM)
- SIMD: All the SPs within an SM execute the same instruction/program in lock step but operate on different data



Cooperation

- Threads operate on different elements of the data but we also need communication
- CUDA allows groups of threads to cooperate through shared memory
- Threads within a block can be barrier synchronized

Programming Model

- CUDA is an extension of C
- Functions that execute on the GPU (device from here onwards) are called *kernels*
- Each running copy of the kernel is called a thread
- Each thread is assigned a unique thread ID so that it may identify itself
- `threadIdx` variable

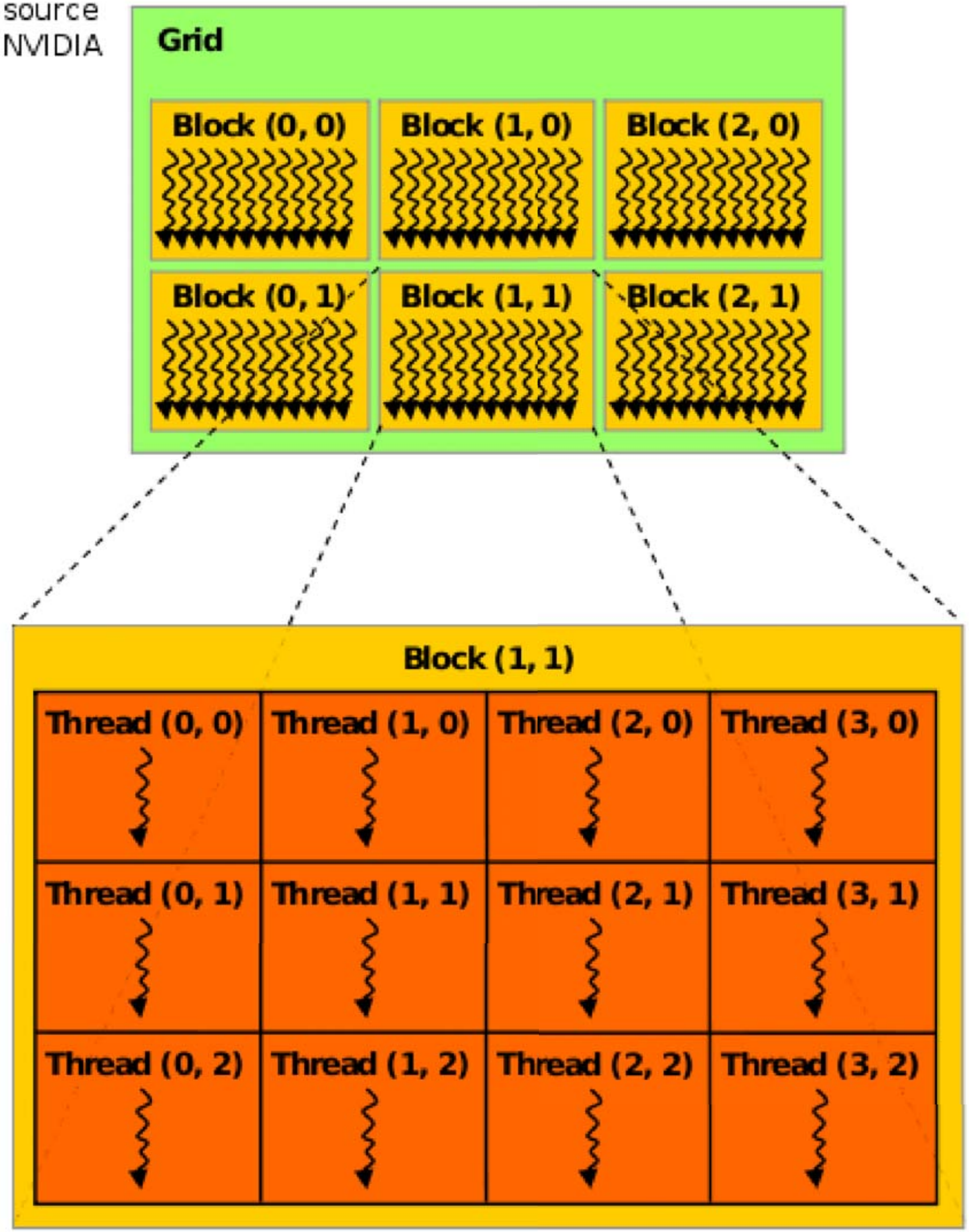
Thread hierarchy

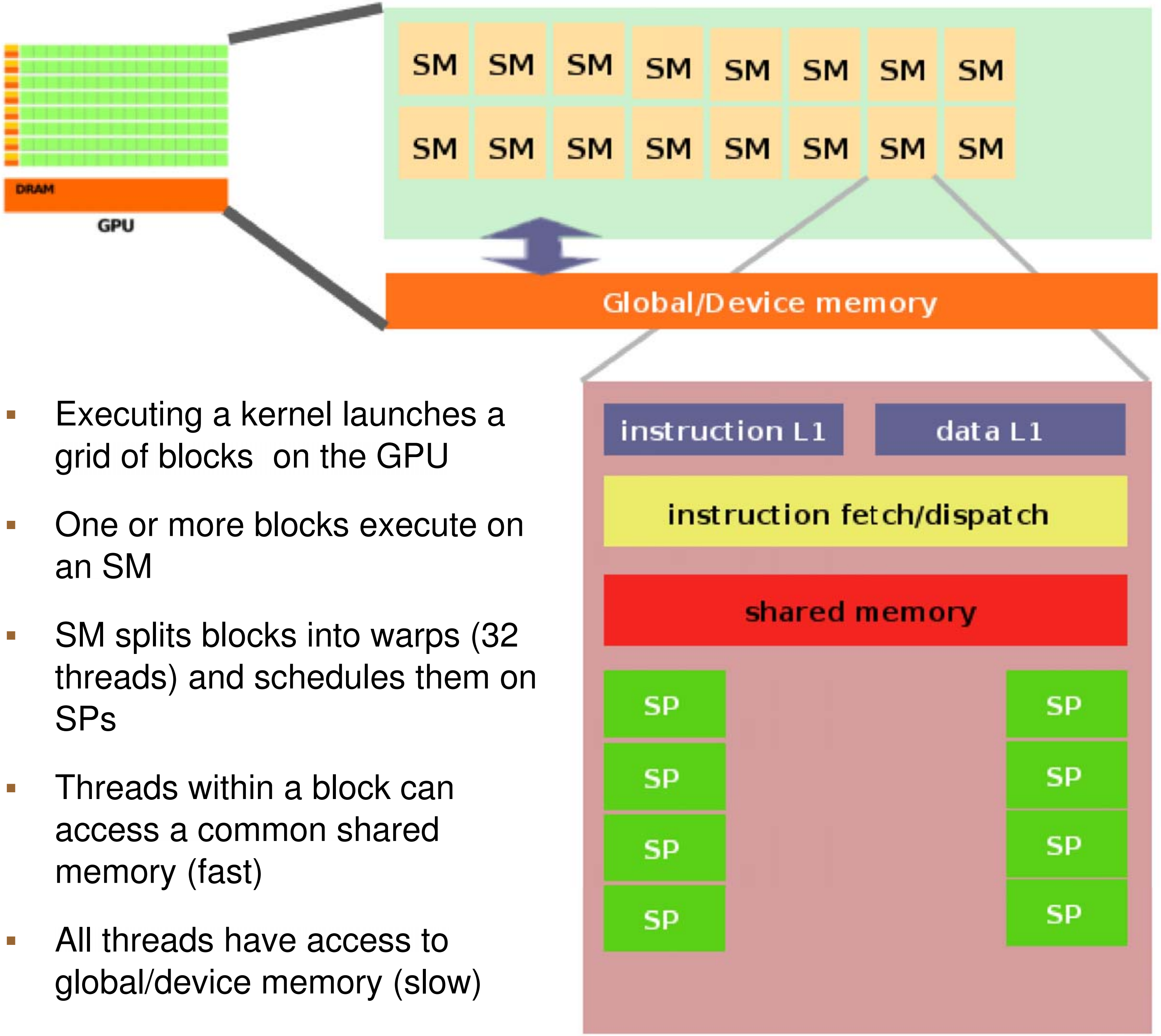
- Threads are grouped into *blocks*
- Blocks can be 1D,2D or 3D
- *`threadIdx.x threadIdx.y threadIdx.z`*
- Thread blocks grouped into 1D or 2D *grid*
- *`blockIdx.x blockIdx.y`*

Thread hierarchy

- Threads can access data based on thread ID
- Example:
  - To index into a one dimensional array using the threadID
  - Within the kernel we can use
    - `array[blockIdx.x * blockDim.x + threadIdx.x]`
  - Since each thread has a unique thread ID, each operates on a unique element in the array
  - Extend this idea to 2D and 3D so that we can logically map threads to elements of matrix data

source
NVIDIA





- Executing a kernel launches a grid of blocks on the GPU
- One or more blocks execute on an SM
- SM splits blocks into warps (32 threads) and schedules them on SPs
- Threads within a block can access a common shared memory (fast)
- All threads have access to global/device memory (slow)

Program structure

- Sequential program on host
- Allocate memory on device
- Copy data to device
- Launch kernel (executes several threads on device. Preferably thousands to occupy the hardware completely)
- Copy results from device

```
int main(int argc, char** argv){
        CUT_DEVICE_INIT(argc,argv);
        int* d_A, * d_B;
        int A[SIZE], B[SIZE];
        cudaMalloc((void**)&d_A,SIZE_BYTES);
        cudaMalloc((void**)&d_B,SIZE_BYTES);
        cudaMemcpy(d_A,A,SIZE_BYTES,cudaMemcpyHostToDevice);
        cudaMemcpy(d_B,B,SIZE_BYTES,cudaMemcpyHostToDevice);
        dim3 threads_in_block(512), blocks(2);
        mult<<<blocks,threads_in_block>>>(d_A,d_B);
        cudaThreadSynchronize();
        cudaError_t error=cudaGetLastError();
        if (error!=cudaSuccess) {
             fprintf(stderr,"Kernel execution failed :
                  %s\n",cudaGetErrorString(error));
                return 1;
        }
        cudaMemcpy(B,d_B,SIZE_BYTES,cudaMemcpyDeviceToHost);
        printf ("Success");
        return 0;
}
```

```
__global__ void mult(int* A, int* B){
        int my=threadIdx.x+blockIdx.x*BlockDim.x;
        A[my]=A[my]+B[my];
}
```

Does it perform well?

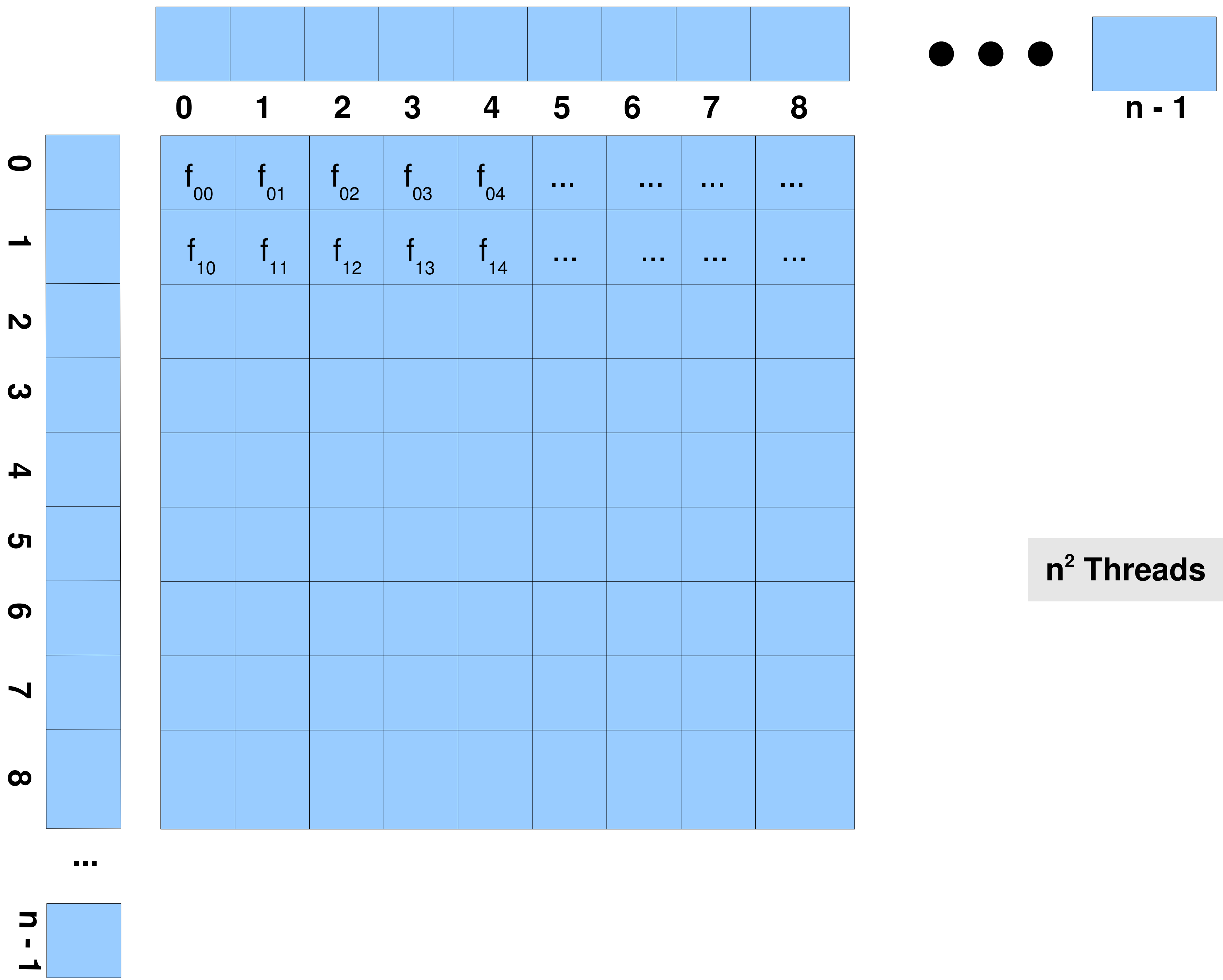
- No.
- To full utilize the hardware we need
  - High arithmetic intensity ie. Number of calculations per device memory access
  - Large number of threads/blocks
  - Device memory accesses have 200 clock cycle latency (very slow)

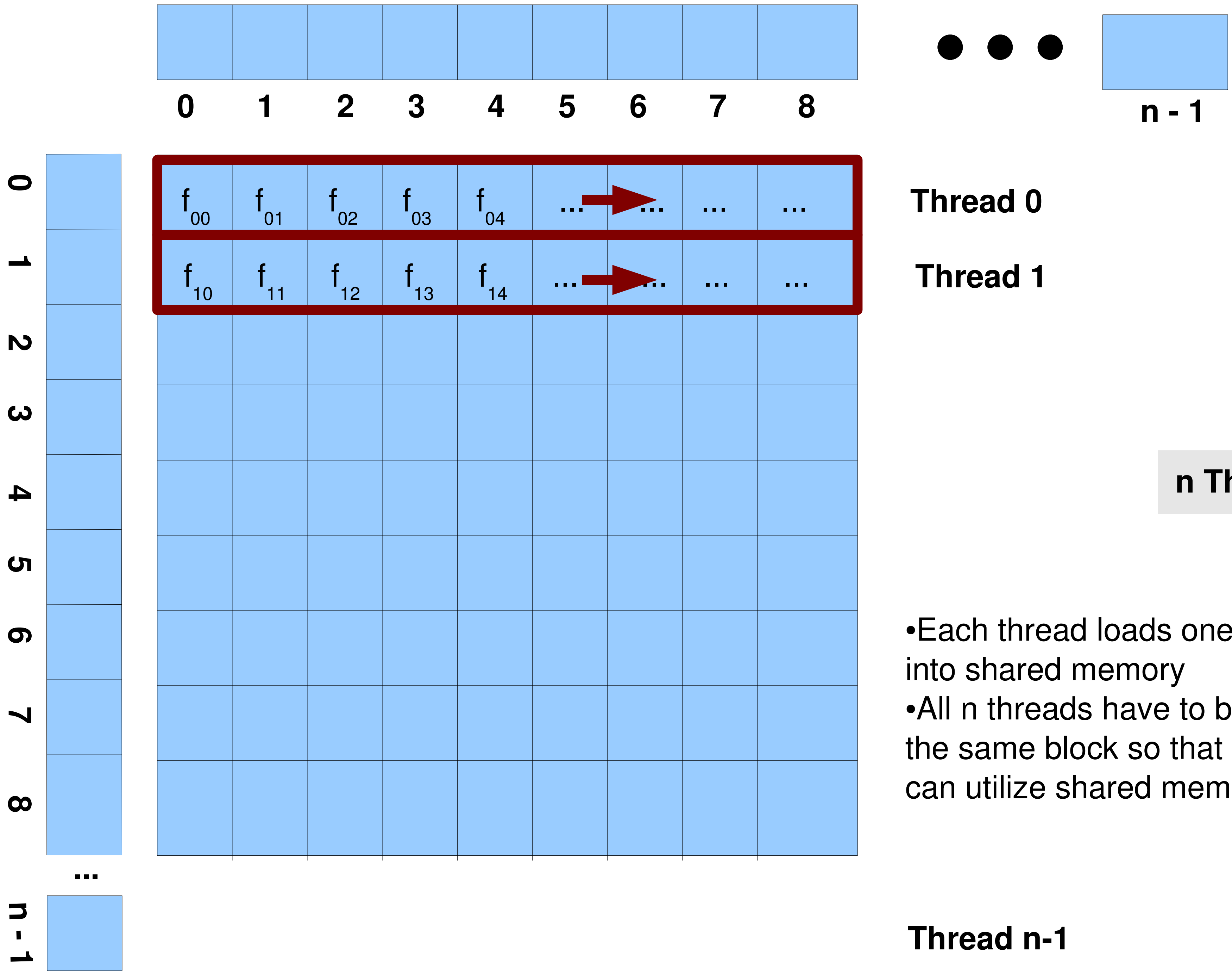
Shared memory

- Shared memory is fast (~2 clock cycles)
- 16KB per SM (16KB available to a thread block)
- Helps us to utilize data reuse for performance by reducing trips to global memory
- We structure computations so that they are performed block-wise

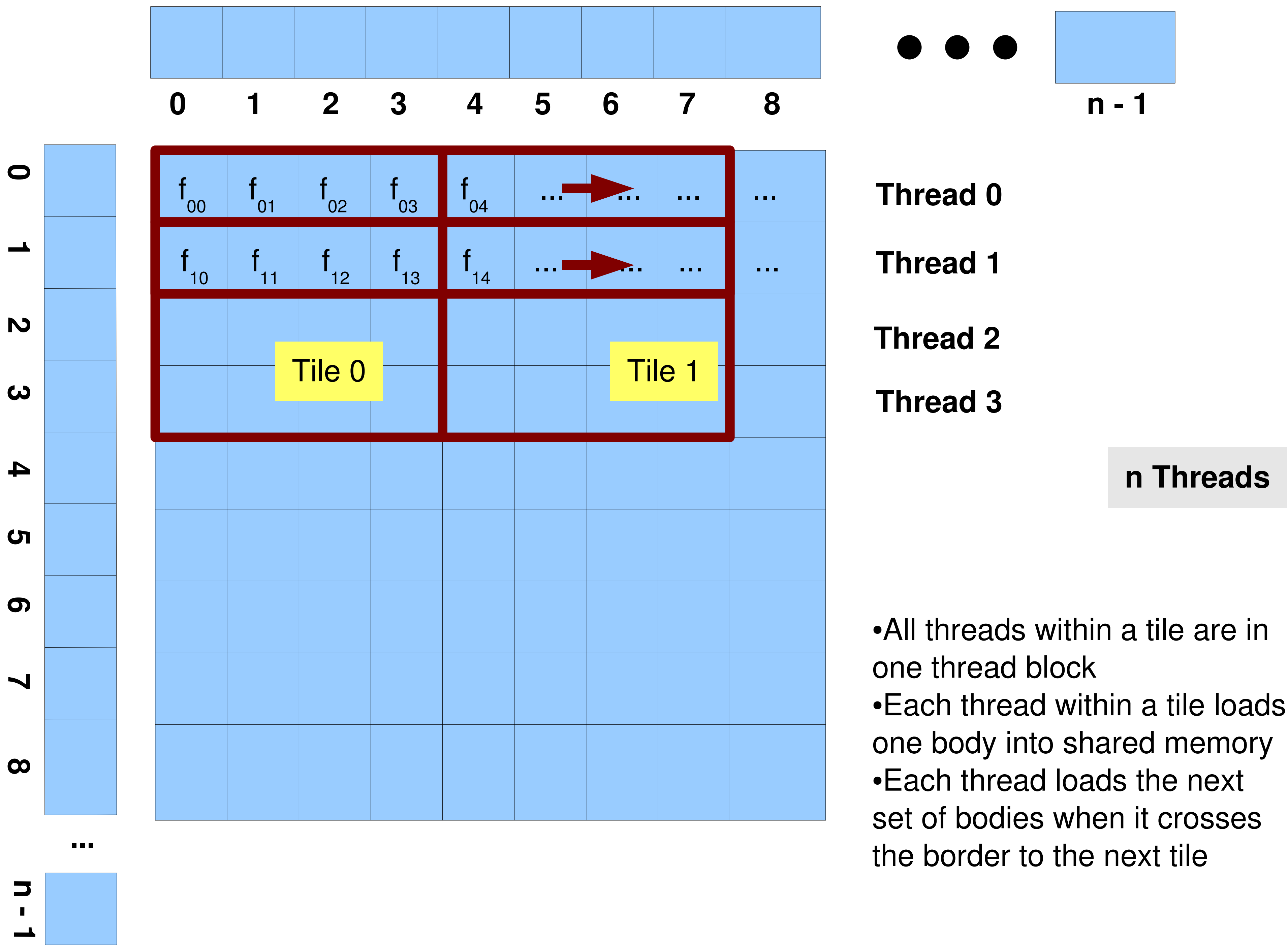
Shared memory

- In an n-body simulation forces are calculated for each body interacting with every other (n-1) bodies
- We represent each body as an element of an n element array


0 1 2 3 4 5 6 7 8
n - 1
f_{00} f_{01} f_{02} f_{03} f_{04} ... ...
f_{10} f_{11} f_{12} f_{13} f_{14} ... ...
0 1 2 3 4 5 6 7 8
...
n - 1
n^2 Threads



- Each thread loads one body into shared memory
- All n threads have to be in the same block so that they can utilize shared memory



- All threads within a tile are in one thread block
- Each thread within a tile loads one body into shared memory
- Each thread loads the next set of bodies when it crosses the border to the next tile

Basic performance tips

- High arithmetic intensity
- Large number of threads
- Each thread loads data from device memory to shared memory
- Process data in shared memory
- Use more blocks

References

- http://www.nvidia.com/cuda
- CUDA classes at University of Illinois video download on NVIDIA CUDA website
  - Taught by Professor Wen-mei W. Hwu and David Kirk, NVIDIA Chief Scientist.
- http://www.gpgpu.org