Multithreaded programming on the GPU: pointers and hints for the computer algebraist

Marc Moreno Maza

University of Western Ontario, London, Ontario
IBM Center for Advanced Studies, Markham, Ontario

PASCO 2017, 23-24 July, Kaisserslautern, Germany

Plan

CUDA: programming, memory and execution models

- CUDA basics
- CUDA programming: more details and examples
- CUDA programming practices

First CUDA programs for the computer algebraist

- Tiled matrix transposition in CUDA
- Tiled matrix multiplication in CUDA
- Something you cannot do on multicores: parallel addition

Analyzing many-core multithreaded algorithms

- A many-core machine model
- Characteristics
- Complexity measures

More CUDA programs for the computer algebraist

- Plain univariate polynomial multiplication
- The Euclidean division
- The Euclidean algorithm

Plan

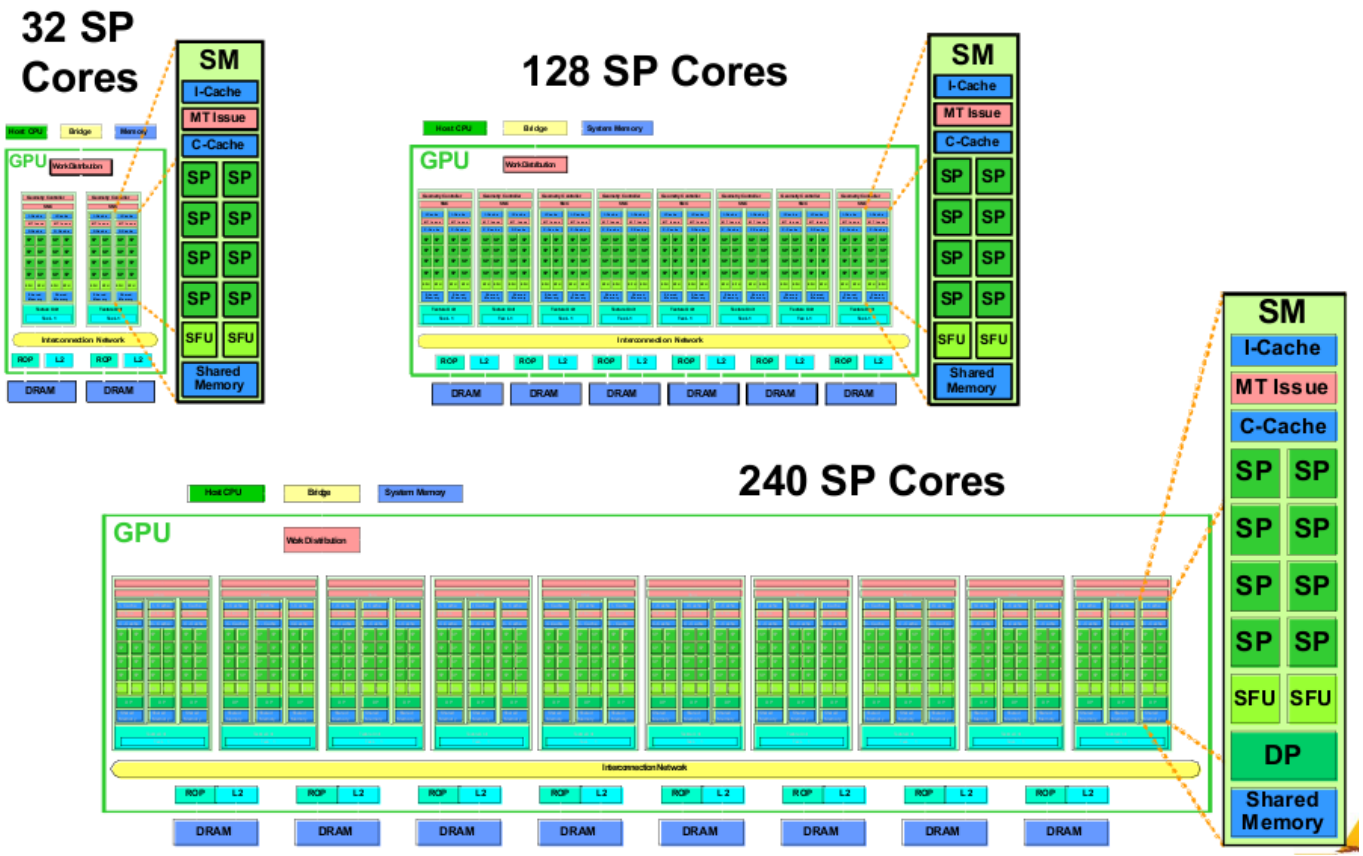
Plan

GPUs

- GPUs are massively multithreaded manycore chips:
  - NVIDIA Tesla products have up to 448 scalar processors with
  - over 12,000 concurrent threads in flight and
  - 1030.4 GFLOPS sustained performance (single precision).
- Users across science & engineering disciplines are achieving 100x or better speedups on GPUs.

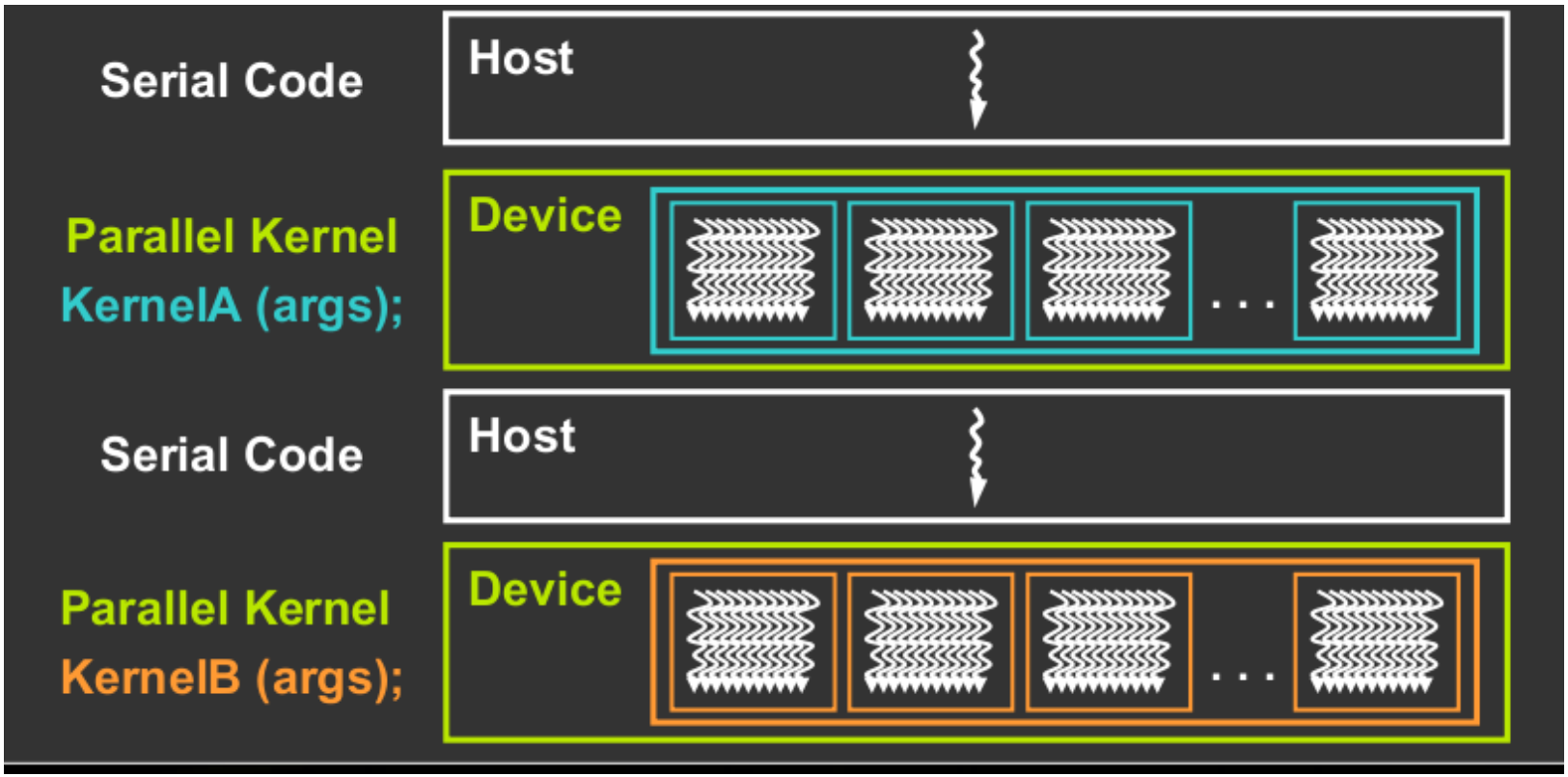
CUDA design goals

- Enable heterogeneous systems (i.e., CPU+GPU)
- Scale to 100's of cores, 1000's of parallel threads
- Use C/C++ with minimal extensions
- Let programmers focus on parallel algorithms



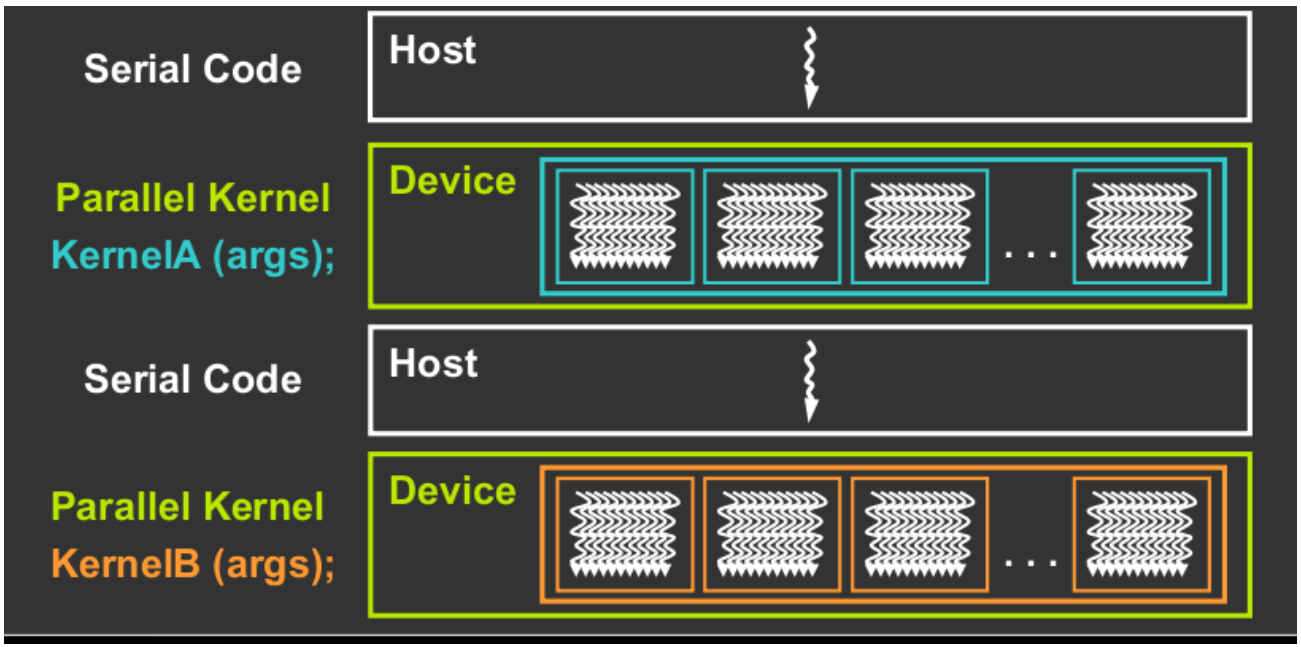
Heterogeneous programming (1/3)

- A CUDA program is a serial program with parallel kernels, all in C.
- The serial C code executes in a host (= CPU) thread
- The parallel kernel C code executes in many device threads across multiple GPU processing elements, called streaming processors (SP).



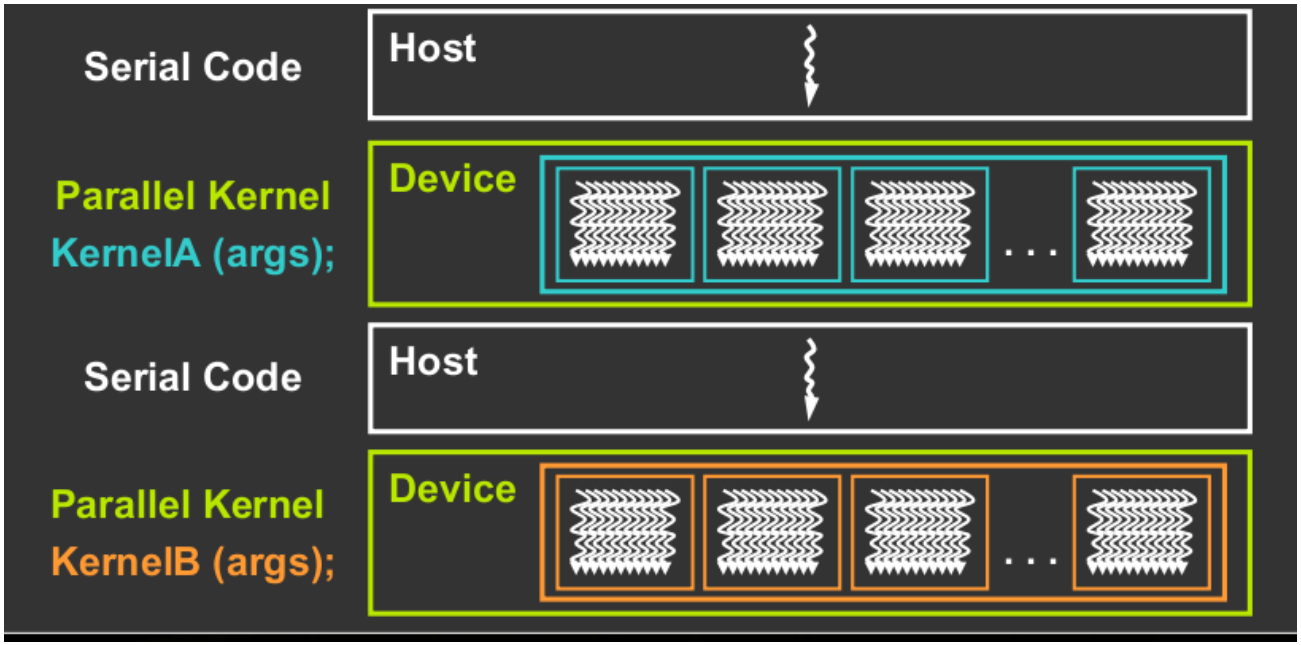
Heterogeneous programming (2/3)

- Thus, the parallel code (kernel) is launched and executed on a device by many threads.
- Threads are grouped into thread blocks.
- One kernel is executed at a time on the device.
- Many threads execute each kernel.



Heterogeneous programming (3/3)

- The parallel code is written for a thread
  - Each thread is free to execute a unique code path
  - Built-in **thread and block ID variables** are used to map each thread to a specific data tile (see next slide).
- Thus, each thread executes the same code on different data based on its thread and block ID.



Example: increment array elements (1/2)

Increment N-element vector a by scalar b

Let's assume N=16, blockDim=4 -> 4 blocks

```
int idx = blockDim.x * blockId.x + threadIdx.x;
```

blockIdx.x=0	blockIdx.x=1	blockIdx.x=2	blockIdx.x=3
blockDim.x=4	blockDim.x=4	blockDim.x=4	blockDim.x=4
threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1
idx=0,1,2,3	idx=4,5,6,7	idx=8,9,10,11	idx=12,13,14,1

See our example number 4 in `/usr/local/cs4402/examples/4`

Example: increment array elements (2/2)

CPU program

```
void increment_cpu(float *a, float b, int N)
{
    for (int idx = 0; idx<N; idx++)
        a[idx] = a[idx] + b;
}


void main()
{
  .....
    increment_cpu(a, b, N);
}
```

CUDA program

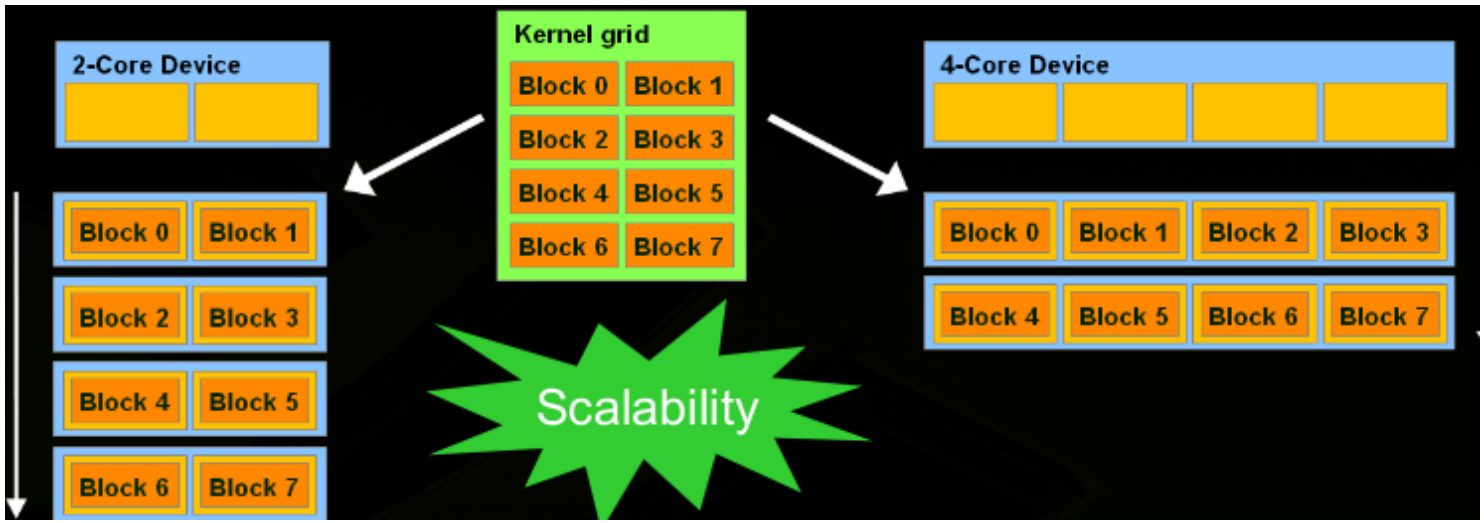
```
__global__ void increment_gpu(float *a, float b, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if( idx < N)
        a[idx] = a[idx] + b;
}


void main()
{
  .....
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float)blocksize)  );
    increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Example host code for increment array elements

```
// allocate host memory
unsigned int numBytes = N * sizeof(float)
float* h_A = (float*) malloc(numBytes);

// allocate device memory
float* d_A = 0;
cudaMalloc((void**)&d_A, numbytes);

// copy data from host to device
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);


// execute the kernel
increment_gpu<<< N/blockSize, blockSize>>>(d_A, b);


// copy data from device back to host
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// free device memory
cudaFree(d_A);
```

Thread blocks (1/2)

- A **Thread block** is a group of threads that can:
  - Synchronize their execution
  - Communicate via shared memory
- Within a grid, **thread blocks can run in any order**:
  - Concurrently or sequentially
  - Facilitates scaling of the same code across many devices



Thread blocks (2/2)

- Thus, within a grid, any possible interleaving of blocks must be valid.
- Thread blocks **may coordinate but not synchronize**
  - they may share pointers
  - they should not share locks (this can easily deadlock).
- The fact that thread blocks cannot synchronize gives **scalability**:
  - A kernel scales across any number of parallel cores
- However, within a thread block, threads may synchronize with barriers.
- That is, threads wait at the barrier until **all** threads in the **same block** reach the barrier.

Vector addition on GPU (1/4)

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run grid of N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Vector addition on GPU (2/4)

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

Host Code

```
int main()
{
    // Run grid of N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Vector addition on GPU (3/4)

```
// allocate and initialize host (CPU) memory
float *h_A = …,    *h_B = …; *h_C = …(empty)

// allocate device (GPU) memory
float *d_A, *d_B, *d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));

// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float),
    cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float),
    cudaMemcpyHostToDevice) );

// execute grid of N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
```

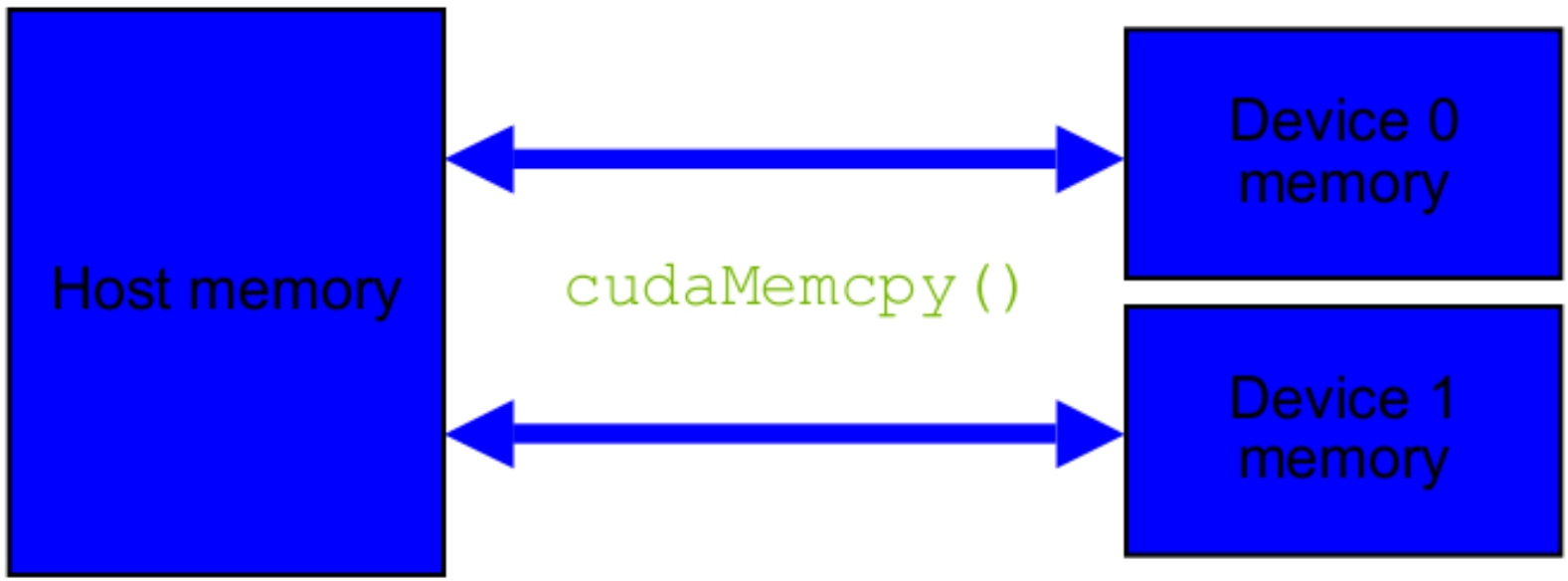
Vector addition on GPU (4/4)

```
// execute grid of N/256 blocks of 256 threads eac
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);

// copy result back to host memory
cudaMemcpy( h_C, d_C, N * sizeof(float),
    cudaMemcpyDeviceToHost) );

// do something with the result...

// free device (GPU) memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Memory hierarchy (1/3)

Host (CPU) memory:

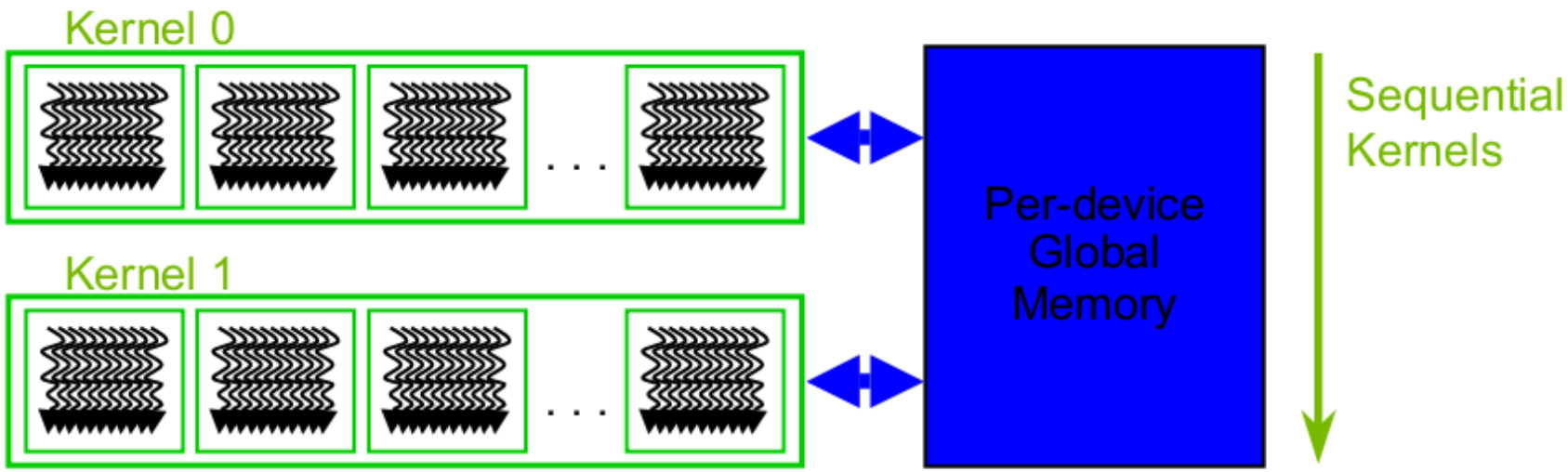
- Not directly accessible by CUDA threads



Memory hierarchy (2/3)

Global (on the device) memory:

- Also called **device memory**
- Accessible by all threads as well as host (CPU)
- Data lifetime = from allocation to deallocation



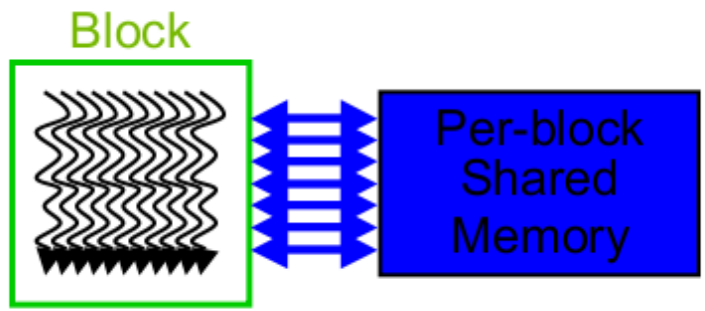
Memory hierarchy (3/3)

Shared memory:

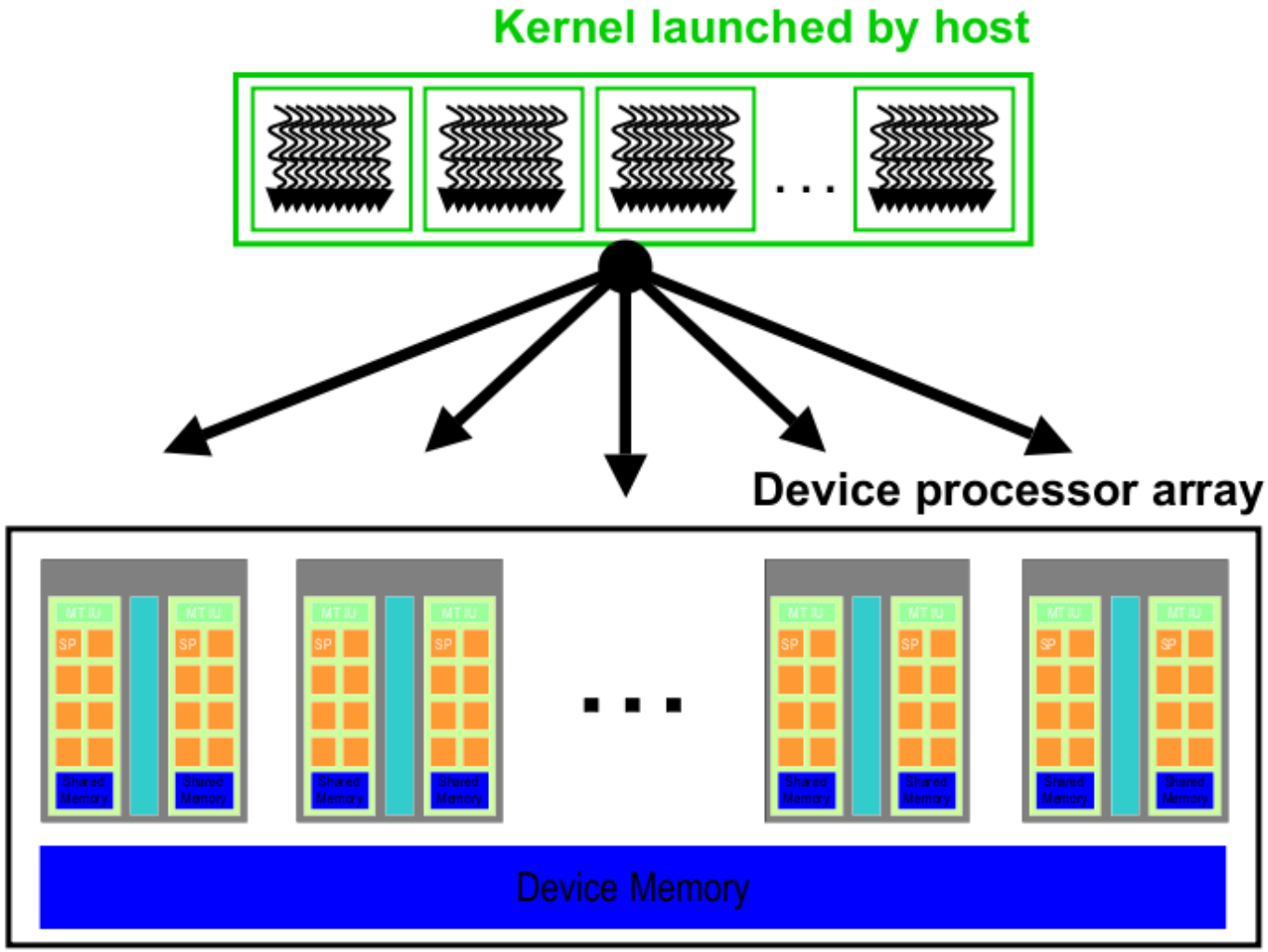
- Each thread block has its own shared memory, which is accessible only by the threads within that block
- Data lifetime = block lifetime

Local storage:

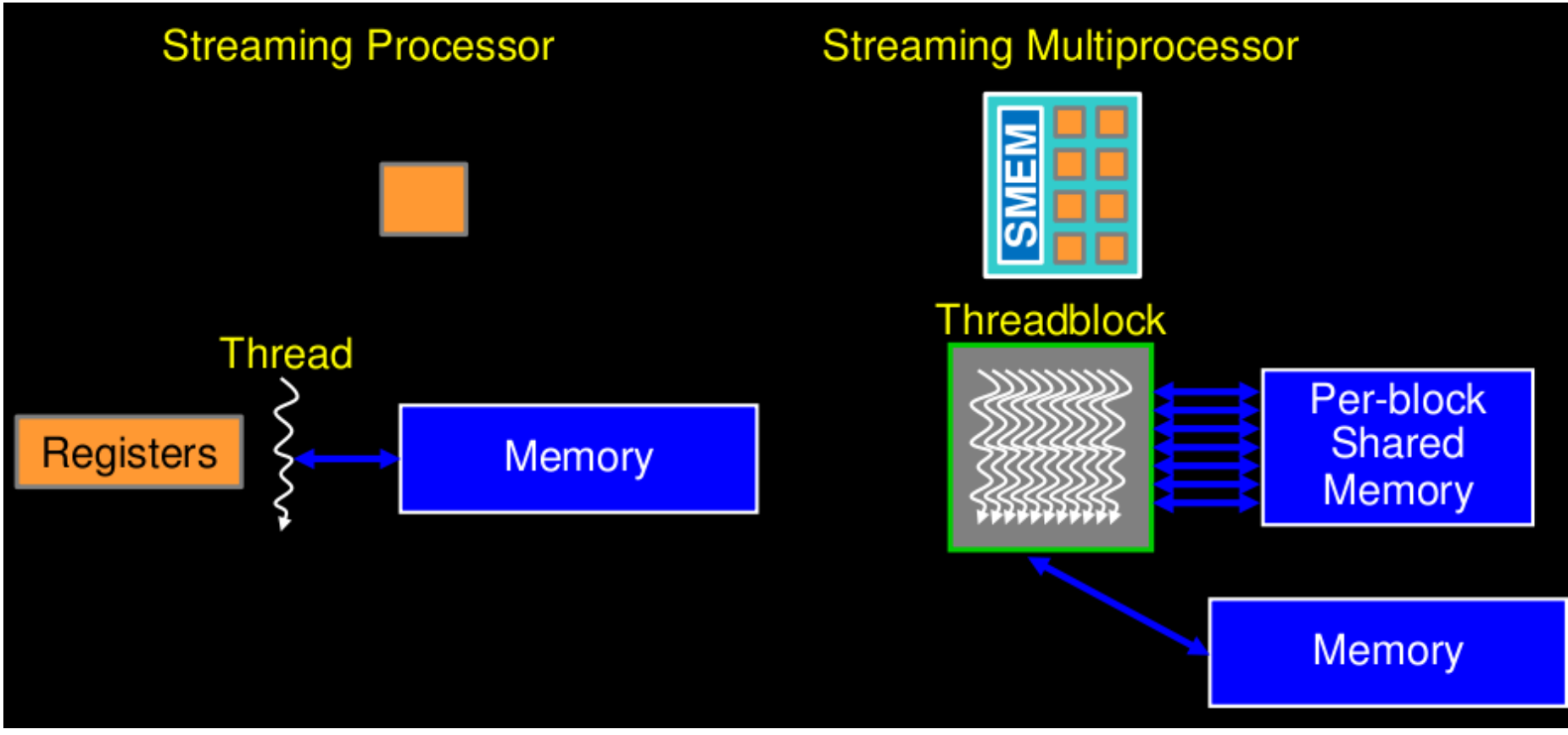
- Each thread has its own local storage
- Data lifetime = thread lifetime



Blocks run on multiprocessors

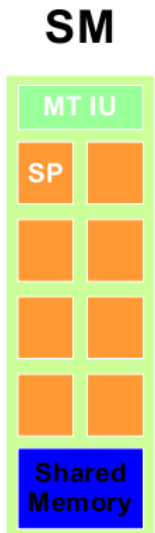


Streaming processors and multiprocessors



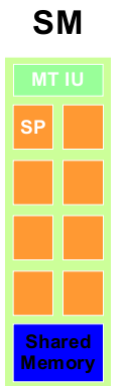
Hardware multithreading

- **Hardware allocates resources to blocks**:
  - blocks need: thread slots, registers, shared memory
  - blocks don't run until resources are available
- **Hardware schedules threads**:
  - threads have their own registers
  - any thread not waiting for something can run
  - context switching is free every cycle
- **Hardware relies on threads to hide latency**:
  - thus high parallelism is necessary for performance.



SIMT thread execution

- At each clock cycle, a multiprocessor executes the same instruction on a group of threads called a **warp**
  - The number of threads in a warp is the **warp size** (32 on G80)
  - A half-warp is the first or second half of a warp.
- Within a warp, threads
  - share instruction fetch/dispatch
  - some become inactive when code path diverges
  - hardware automatically handles divergence
- **Warps are the primitive unit of scheduling**:
  - each active block is split into warps in a well-defined way
  - threads within a warp are executed physically in parallel while warps and blocks are executed logically in parallel.



Plan

Code executed on the GPU

- The GPU code defines and calls C function with some restrictions:
  - Can only access GPU memory
  - No variable number of arguments
  - No static variables
  - No recursion (...well this has changed recently)
  - No dynamic polymorphism

- GPU functions must be declared with a qualifier:

  `__global__` : launched by CPU, cannot be called from GPU, must return void

  `__device__` : called from other GPU functions, cannot be launched by the CPU

  `__host__` : can be executed by CPU

- qualifiers can be combined.

- Built-in variables: `gridDim`, `blockDim`, `blockIdx`, `threadIdx`

Variable qualifiers (GPU code)

`__device__` :
- stored in global memory (not cached, high latency)
- accessible by all threads
- lifetime: application

`__constant__` :
- stored in global memory (cached)
- read-only for threads, written by host
- Lifetime: application

`__shared__` :
- stored in shared memory (latency comparable to registers)
- accessible by all threads in the same threadblock
- lifetime: block lifetime

Unqualified variables:
- scalars and built-in vector types are stored in registers
- arrays are stored in device (= global) memory

Launching kernels on GPU

Launch parameters:

- grid dimensions (up to 2D)
- thread-block dimensions (up to 3D)
- shared memory: number of bytes per block
  - for extern smem variables declared without size
  - optional, 0 by default
- stream ID:
  - Optional, 0 by default

```
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block, 0, 0>>>(...);
kernel<<<32, 512>>>(...);
```

GPU memory allocation / release

Host (CPU) manages GPU memory:

- `cudaMalloc (void ** pointer, size_t nbytes)`
- `cudaMemset (void * pointer, int value, size_t count)`
- `cudaFree (void* pointer)`

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int * d_a = 0;
cudaMalloc( (void**)&d_a, nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Data copies

- `cudaMemcpy( void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);`
  - returns after the copy is complete,
  - blocks the CPU thread,
  - doesn't start copying until previous CUDA calls complete.
- `enum cudaMemcpyKind`
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToHost`
  - `cudaMemcpyDeviceToDevice`
- Non-blocking memcopies are provided (more on this later)

Thread synchronization function

- `void __syncthreads();`
- Synchronizes all threads in a block:
  - once all threads have reached this point, execution resumes normally.
  - this is used to avoid hazards when accessing shared memory.
- Should be used in conditional code only if the condition is uniform across the entire thread block.

Kernel variations and output: what is in a?

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}
```

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = blockIdx.x;
}
```

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = threadIdx.x;
}
```

Kernel variations and utput: answers

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}
```

Output: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = blockIdx.x;
}
```

Output: 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3

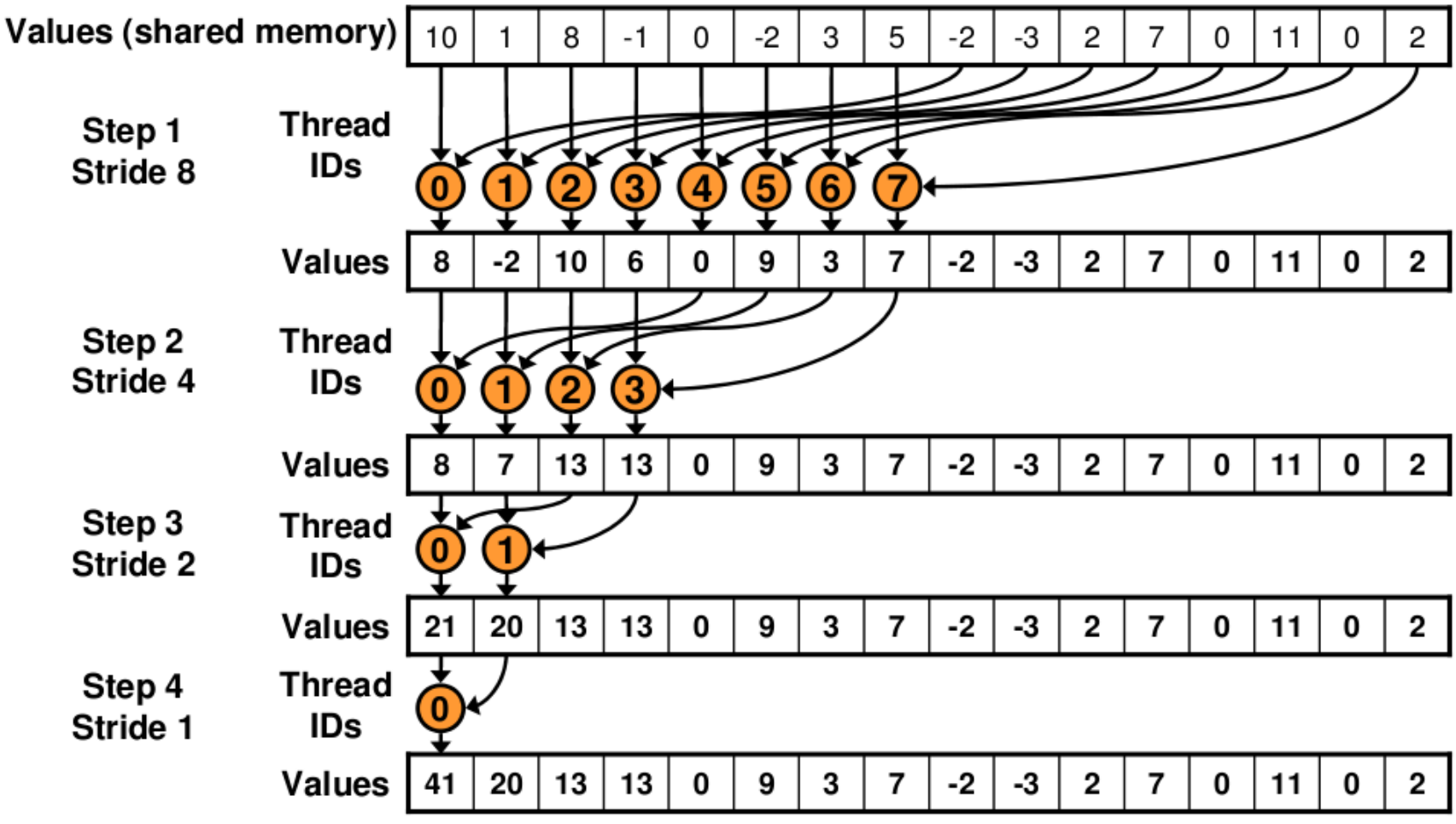
```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = threadIdx.x;
}
```

Output: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

Example kernel source code: what does this do?

```
__global__ void sum_kernel(int *g_input, int *g_output)
{
    extern __shared__ int s_data[ ];  // allocated during kernel launch

    // read input into shared memory
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    s_data[ threadIdx.x ] = g_input[ idx ];
    __syncthreads( );

    // compute sum for the threadblock
    for ( int dist = blockDim.x/2;  dist > 0;  dist /= 2 )
    {
        if ( threadIdx.x < dist )
            s_data[ threadIdx.x ] += s_data[ threadIdx.x + dist ];
        __syncthreads( );
    }

    // write the block's sum to global memory
    if ( threadIdx.x == 0 )
        g_output[ blockIdx.x ] = s_data[0];
}
```

Example kernel source code: solution



Sequential addressing is conflict free

Kernel with 2D Indexing (1/2)

```
__global__ void kernel( int *a, int dimx, int dimy )
{
    int ix  = blockIdx.x*blockDim.x + threadIdx.x;
    int iy  = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;

    a[idx] = a[idx]+1;
}
```

Kernel with 2D Indexing (2/2)

```
__global__ void kernel( int *a, int dimx, int dimy )
{
  int ix  = blockIdx.x*blockDim.x + threadIdx.x;
  int iy  = blockIdx.y*blockDim.y + threadIdx.y;
  int idx = iy*dimx + ix;

  a[idx] = a[idx]+1;
}
```

```
int main()
{
  int dimx = 16;
  int dimy = 16;
  int num_bytes = dimx*dimy*sizeof(int);

  int *d_a=0, *h_a=0; // device and host pointers

  h_a = (int*)malloc(num_bytes);
  cudaMalloc( (void**)&d_a, num_bytes );

  if( 0==h_a || 0==d_a )
  {
    printf("couldn't allocate memory\n");
    return 1;
  }

  cudaMemset( d_a, 0, num_bytes );

  dim3 grid, block;
  block.x = 4;
  block.y = 4;
  grid.x  = dimx / block.x;
  grid.y  = dimy / block.y;

  kernel<<<grid, block>>>( d_a, dimx, dimy );

  cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

  for(int row=0; row<dimy; row++)
  {
    for(int col=0; col<dimx; col++)
      printf("%d ", h_a[row*dimx+col] );
    printf("\n");
  }

  free( h_a );
  cudaFree( d_a );

  return 0;
}
```

Plan

Four principles

- Expose as much parallelism as possible
  - If threads of same block need to communicate, use shared memory and `__syncthreads()`
  - If threads of different blocks need to communicate, use global memory and split computation into multiple kernels
  - High parallelism is especially important to hide memory latency by overlapping memory accesses with computation
- Optimize memory usage for maximum bandwidth
  - Effective bandwidth can vary by an order of magnitude
  - Optimize access patterns to get:
    - Coalesced global memory accesses, and
    - Shared memory accesses with no or few bank conflicts.
- Maximize occupancy to hide latency
  - Leverage parallelism to hide memory latency by overlapping memory accesses with computation as much as possible
  - Sometimes recompute data rather than cache it
  - Write kernels with high arithmetic intensity
- Optimize instruction usage for maximum throughput
  - For instance some 32-bit instructions may yield better throughput than 64-bit counterpart instructions.

Four principles

- Expose as much parallelism as possible
  - If threads of same block need to communicate, use shared memory and `__syncthreads()`
  - If threads of different blocks need to communicate, use global memory and split computation into multiple kernels
  - High parallelism is especially important to hide memory latency by overlapping memory accesses with computation
- Optimize memory usage for maximum bandwidth
  - Effective bandwidth can vary by an order of magnitude
  - Optimize access patterns to get:
    - Coalesced global memory accesses, and
    - Shared memory accesses with no or few bank conflicts.
- Maximize occupancy to hide latency
  - Leverage parallelism to hide memory latency by overlapping memory accesses with computation as much as possible
  - Sometimes recompute data rather than cache it
  - Write kernels with high arithmetic intensity
- Optimize instruction usage for maximum throughput
  - For instance some 32-bit instructions may yield better throughput than 64-bit counterpart instructions.

Four principles

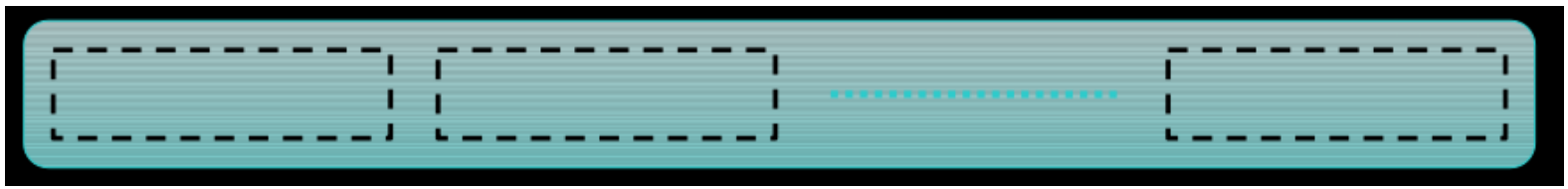
- Expose as much parallelism as possible
  - If threads of same block need to communicate, use shared memory and `__syncthreads()`
  - If threads of different blocks need to communicate, use global memory and split computation into multiple kernels
  - High parallelism is especially important to hide memory latency by overlapping memory accesses with computation
- Optimize memory usage for maximum bandwidth
  - Effective bandwidth can vary by an order of magnitude
  - Optimize access patterns to get:
    - Coalesced global memory accesses, and
    - Shared memory accesses with no or few bank conflicts.
- Maximize occupancy to hide latency
  - Leverage parallelism to hide memory latency by overlapping memory accesses with computation as much as possible
  - Sometimes recompute data rather than cache it
  - Write kernels with high arithmetic intensity
- Optimize instruction usage for maximum throughput
  - For instance some 32-bit instructions may yield better throughput than 64-bit counterpart instructions.

Four principles

- Expose as much parallelism as possible
  - If threads of same block need to communicate, use shared memory and `__syncthreads()`
  - If threads of different blocks need to communicate, use global memory and split computation into multiple kernels
  - High parallelism is especially important to hide memory latency by overlapping memory accesses with computation
- Optimize memory usage for maximum bandwidth
  - Effective bandwidth can vary by an order of magnitude
  - Optimize access patterns to get:
    - Coalesced global memory accesses, and
    - Shared memory accesses with no or few bank conflicts.
- Maximize occupancy to hide latency
  - Leverage parallelism to hide memory latency by overlapping memory accesses with computation as much as possible
  - Sometimes recompute data rather than cache it
  - Write kernels with high arithmetic intensity
- Optimize instruction usage for maximum throughput
  - For instance some 32-bit instructions may yield better throughput than 64-bit counterpart instructions.

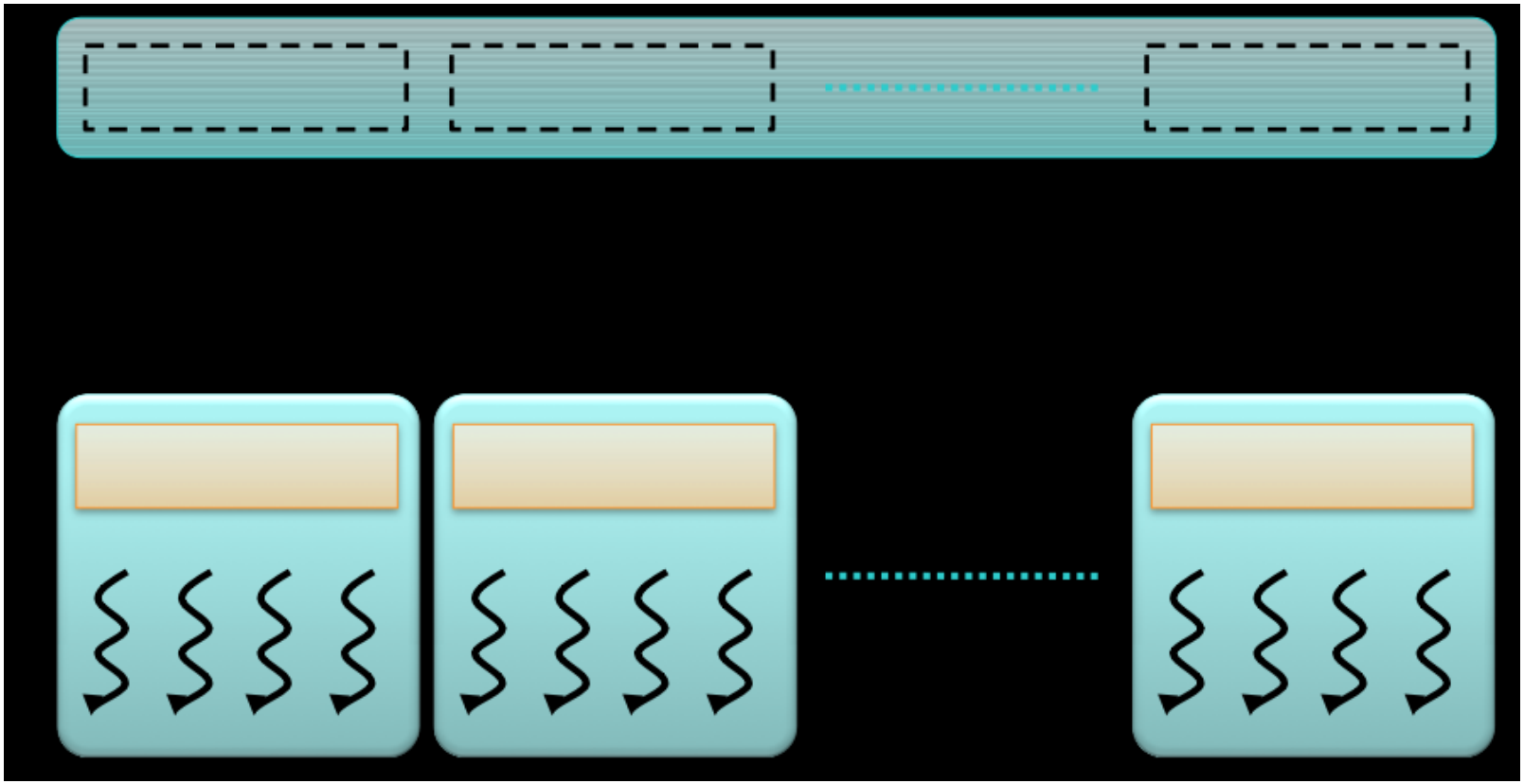
A popular programming strategy (1/5)

Partition data into subsets that fit into shared memory



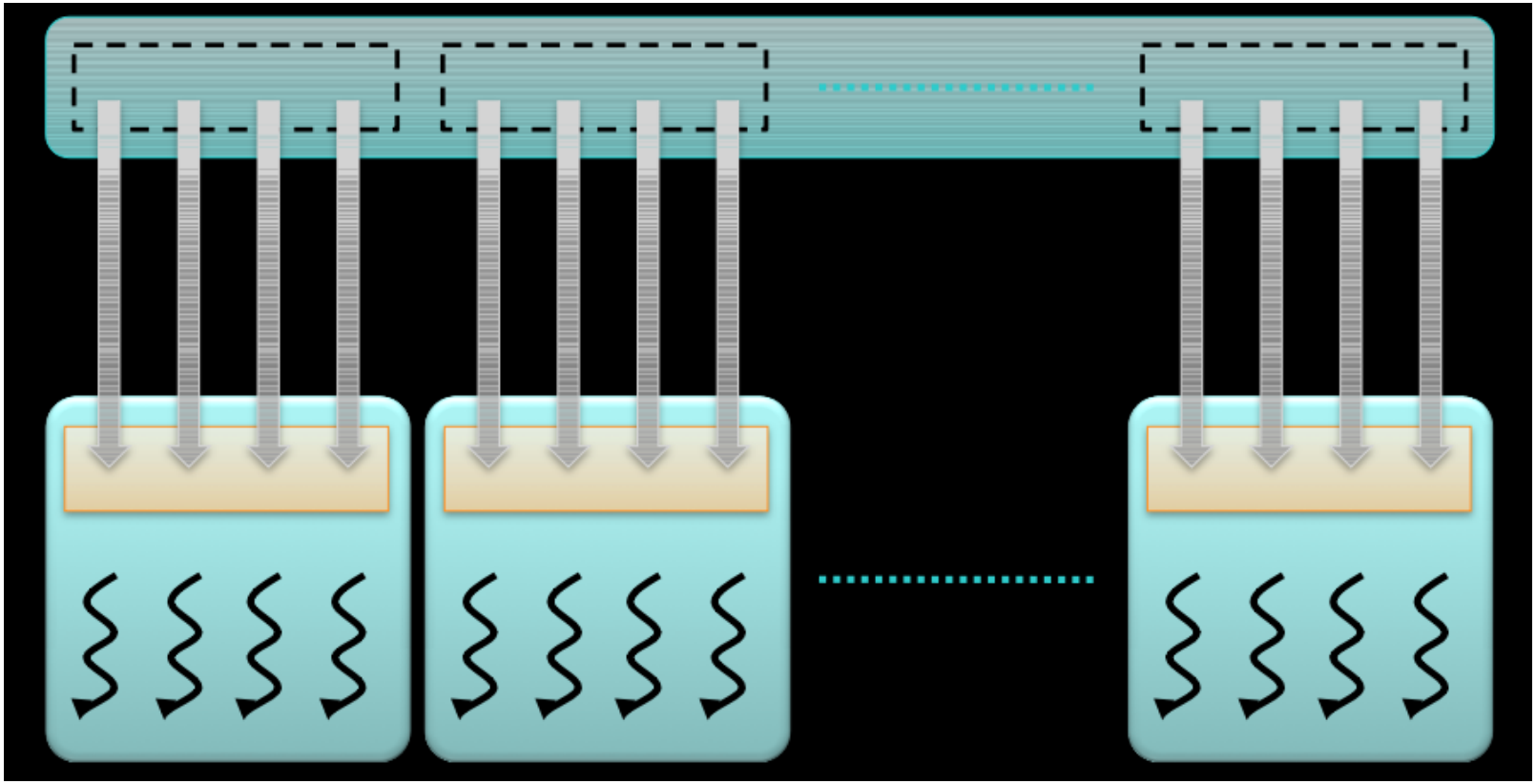
A popular programming strategy (2/5)

Handle each data subset with one thread block



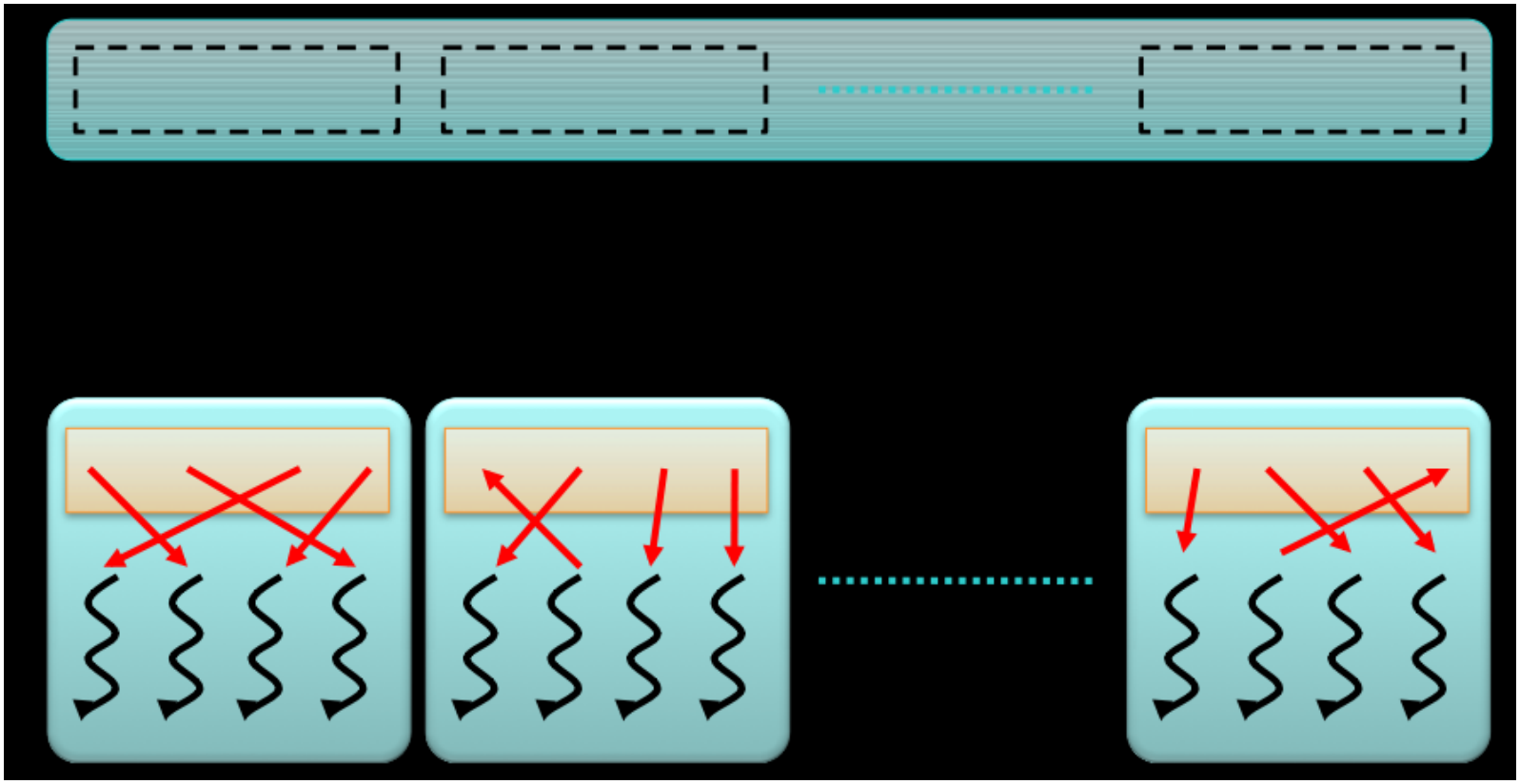
A popular programming strategy (3/5)

Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism.



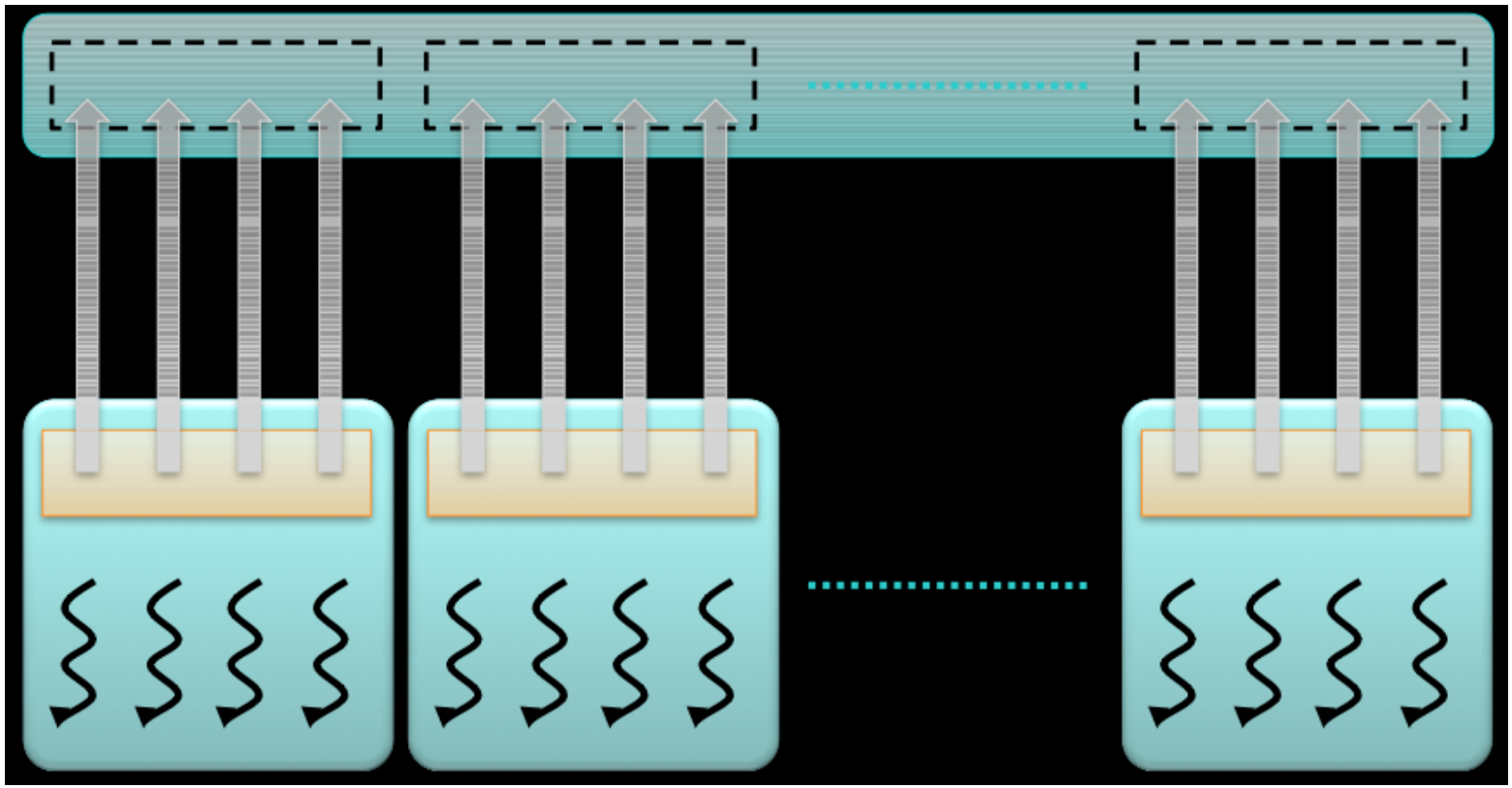
A popular programming strategy (4/5)

Perform the computation on the subset from shared memory.



A popular programming strategy (5/5)

Copy the result from shared memory back to global memory.



Plan

Plan

Matrix transpose characteristics (1/2)

- We optimize a transposition code for a matrix of floats. This operates out-of-place:
  - input and output matrices address separate memory locations.
- For simplicity, we consider an $n \times n$ matrix where 32 divides n.
- We focus on the device code:
  - the host code performs typical tasks: data allocation and transfer between host and device, the launching and timing of several kernels, result validation, and the deallocation of host and device memory.
- Benchmarks illustrate this section:
  - we compare our **matrix transpose** kernels against a **matrix copy** kernel,
  - for each kernel, we compute the **effective bandwidth**, calculated in GB/s as twice the size of the matrix (once for reading the matrix and once for writing) divided by the time of execution,
  - Each operation is run `NUM_REFS` times (for **normalizing the measurements**),
  - This looping is performed **once over the kernel** and once **within the kernel**,
  - The difference between these two timings is kernel launch and synchronization overheads.

Matrix transpose characteristics (2/2)

- We present hereafter different kernels called from the host code, each addressing different performance issues.
- All kernels in this study launch thread blocks of dimension 32x8, where each block transposes (or copies) a tile of dimension 32x32.
- As such, the parameters `TILE_DIM` and `BLOCK_ROWS` are set to 32 and 8, respectively.
- Using a thread block with fewer threads than elements in a tile is advantageous for the matrix transpose:
  - each thread transposes several matrix elements, four in our case, and much of the cost of calculating the indices is amortized over these elements.
- This study is based on a technical report by Greg Ruetsch (NVIDIA) and Paulius Micikevicius (NVIDIA).

A simple copy kernel (1/2)

```
__global__ void copy(float *odata, float* idata, int width,
                                                 int height, int nreps)
{
  int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;
  int index  = xIndex + width*yIndex;

  for (int r=0; r < nreps; r++) { // normalization outer loop
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      odata[index+i*width] = idata[index+i*width];
    }
  }
}
```

A simple copy kernel (2/2)

- `odata` and `idata` are pointers to the input and output matrices,
- `width` and `height` are the matrix x and y dimensions,
- `nreps` determines how many times the loop over data movement between matrices is performed.
- In this kernel, `xIndex` and `yIndex` are global 2D matrix indices,
- used to calculate `index`, the 1D index used to access matrix elements.

```
__global__ void copy(float *odata, float* idata, int width,
                                                 int height, int nreps)
{
  int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;
  int index  = xIndex + width*yIndex;

  for (int r=0; r < nreps; r++) {
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      odata[index+i*width] = idata[index+i*width];
    } } }
```

A naive transpose kernel

```
  _global__ void transposeNaive(float *odata, float* idata,
                                int width, int height, int nreps)
{
   int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
   int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;
   int index_in = xIndex + width * yIndex;
   int index_out = yIndex + height * xIndex;
   for (int r=0; r < nreps; r++) {
     for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
       odata[index_out+i] = idata[index_in+i*width];
     }
   }
}
```

Naive transpose kernel vs copy kernel

The performance of these two kernels on a 2048x2048 matrix using a GTX280 is given in the following table:

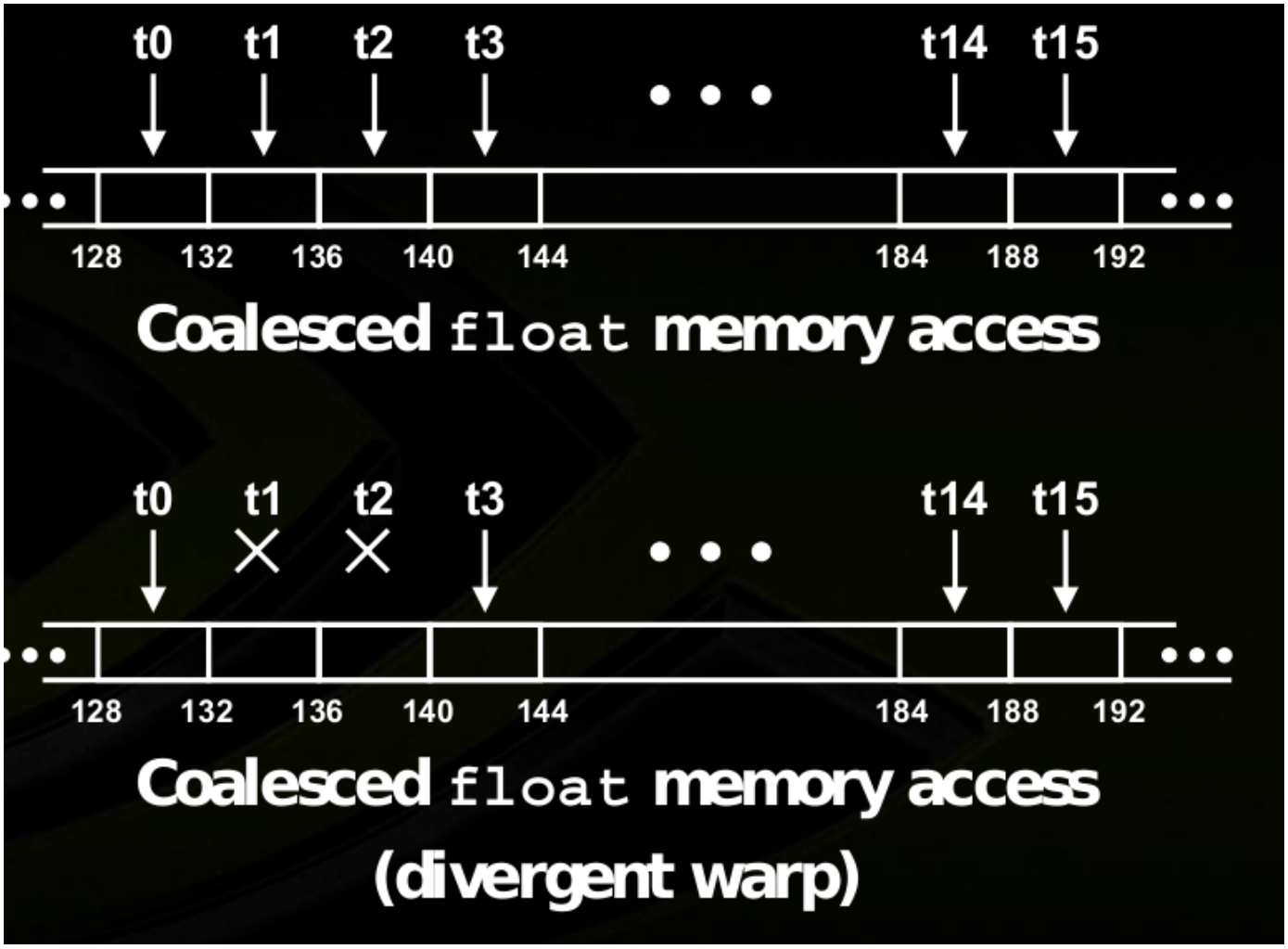
Routine	Bandwidth (GB/s)
copy	105.14
naive transpose	18.82

The minor differences in code between the copy and nave transpose kernels have a profound effect on performance.

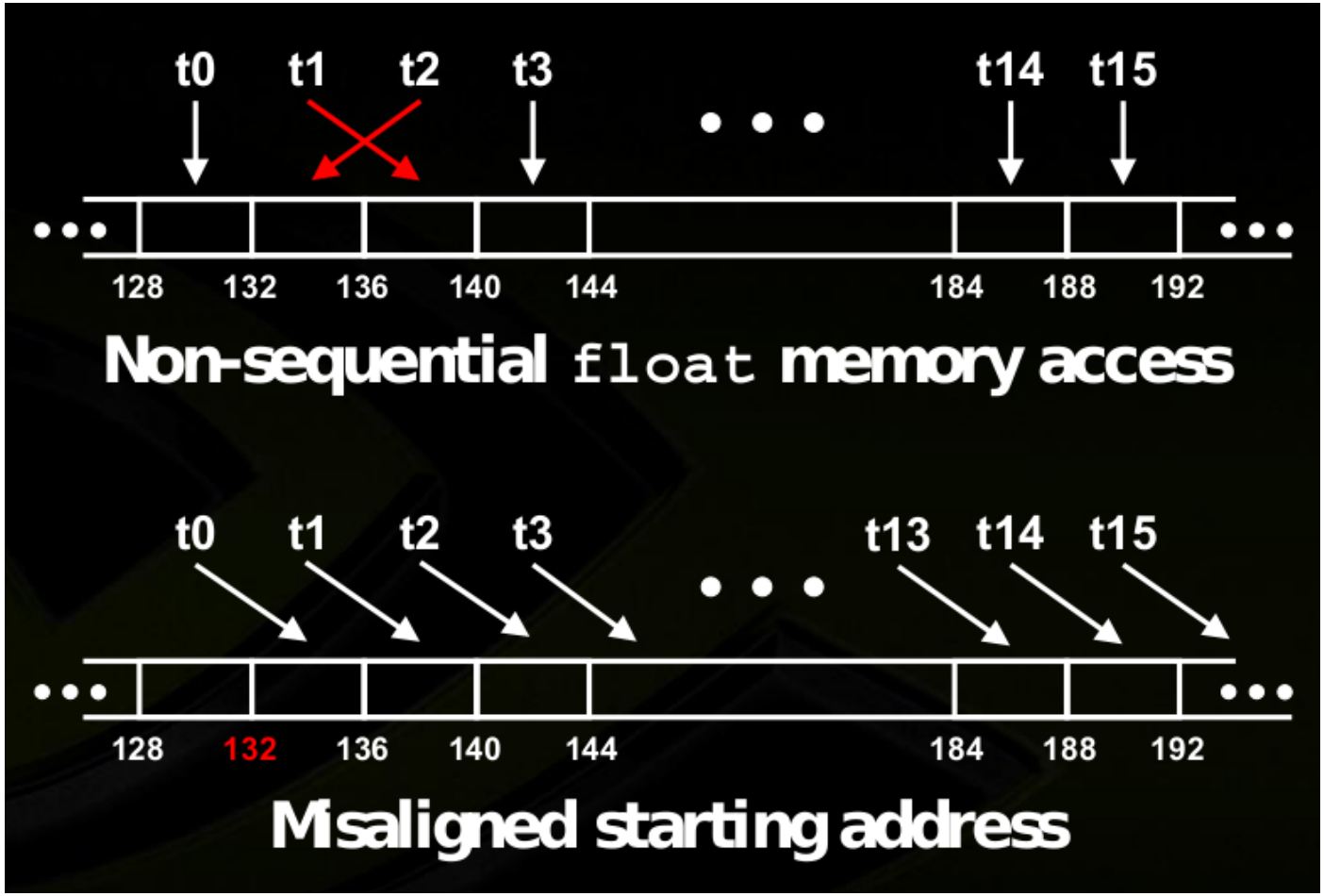
Coalesced Transpose (1/10)

- Because device memory has a much higher latency and lower bandwidth than on-chip memory, special attention must be paid to: **how global memory accesses are performed?**
- The simultaneous global memory accesses by each thread of a half-warp (16 threads on G80) during the execution of a single read or write instruction will be **coalesced** into a single access if:
  1. The size of the memory element accessed by each thread is either 4, 8, or 16 bytes.
  2. The address of the first element is aligned to 16 times the element's size.
  3. The elements form a contiguous block of memory.
  4. The i-th element is accessed by the i-th thread in the half-warp.
- Last two requirements are relaxed with compute capabilities of 1.2.
- Coalescing happens even if some threads do not access memory (**divergent warp**)

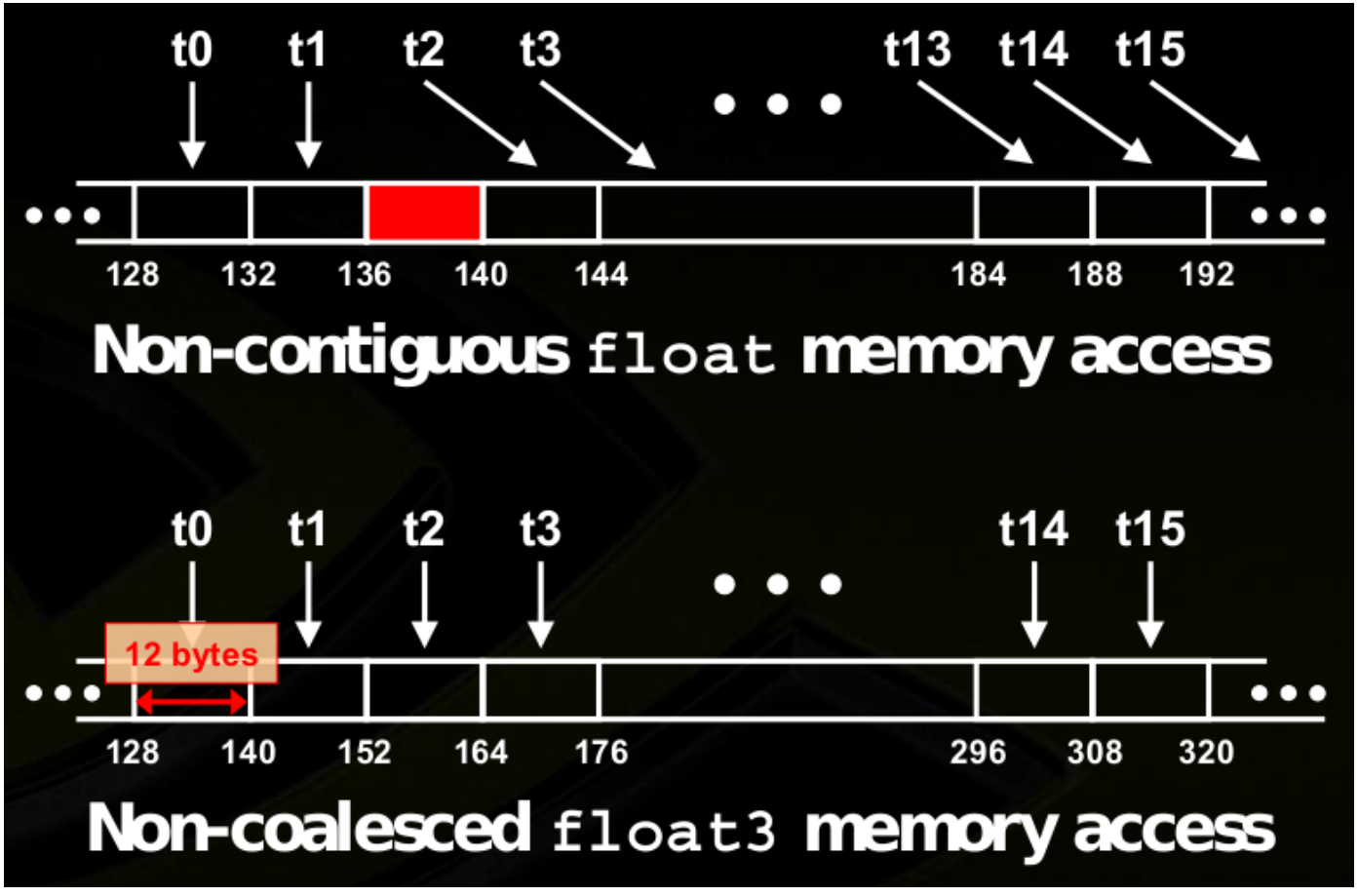
Coalesced Transpose (2/10)



Coalesced Transpose (3/10)



Coalesced Transpose (4/10)



Coalesced Transpose (5/10)

- **Allocating device memory through cudaMalloc()** and choosing `TILE_DIM` **to be a multiple of 16 ensures alignment** with a segment of memory, therefore all loads from `idata` are coalesced.
- Coalescing behavior differs between the simple copy and naive transpose kernels when writing to `odata`.
- In the case of the naive transpose, for each iteration of the `i`-loop a half warp writes one half of a column of floats to different segments of memory:
  - resulting in 16 separate memory transactions,
  - regardless of the compute capability.

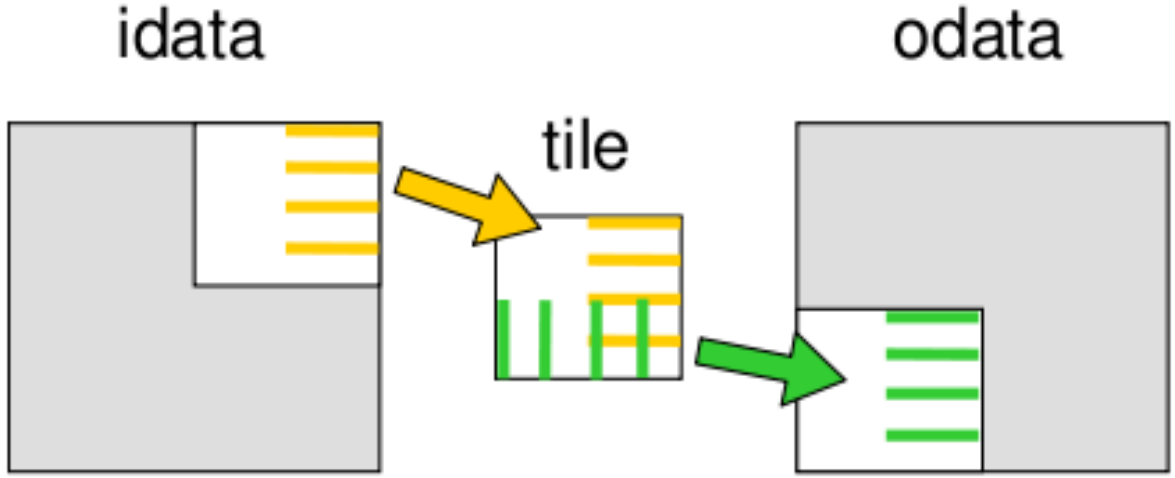
Coalesced Transpose (6/10)

- The way to avoid uncoalesced global memory access is
  1. to read the data into shared memory and,
  2. have each half warp access non-contiguous locations in shared memory in order to write contiguous data to `odata`.
- There is no performance penalty for non-contiguous access patterns in shared memory as there is in global memory.
- a `__synchthreads()` call is required to ensure that all reads from `idata` to shared memory have completed before writes from shared memory to `odata` commence.

Coalesced Transpose (7/10)

```
__global__ void transposeCoalesced(float *odata,
           float *idata, int width, int height) // no nreps param
{
  __shared__ float tile[TILE_DIM][TILE_DIM];
  int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;
  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*height;
  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      tile[threadIdx.y+i][threadIdx.x] =
        idata[index_in+i*width];
  }   __syncthreads();
  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      odata[index_out+i*height] =
        tile[threadIdx.x][threadIdx.y+i];
} }
```

Coalesced Transpose (8/10)



1. The half warp writes four half rows of the idata matrix tile to the shared memory 32x32 array `tile` indicated by the yellow line segments.
2. After a `__syncthreads()` call to ensure all writes to tile are completed,
3. the half warp writes four half columns of tile to four half rows of an odata matrix tile, indicated by the green line segments.

Coalesced Transpose (9/10)

```
__global__ void copySharedMem(float *odata, float *idata,
                              int width, int height) // no nreps pa
{
  __shared__ float tile[TILE_DIM][TILE_DIM];
  int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;
  int index = xIndex + width*yIndex;
  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      tile[threadIdx.y+i][threadIdx.x] =
        idata[index+i*width];
  }
  __syncthreads();
  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      odata[index+i*width] =
        tile[threadIdx.y+i][threadIdx.x];
} }
```

Coalesced Transpose (10/10)

Routine	Bandwidth (GB/s)
copy	105.14
shared memory copy	104.49
naive transpose	18.82
coalesced transpose	51.42

The shared memory copy results seem to suggest that the use of shared memory with a synchronization barrier has little effect on the performance, certainly as far as the *Loop in kernel* column indicates when comparing the simple copy and shared memory copy.

Shared memory bank conflicts (1/6)

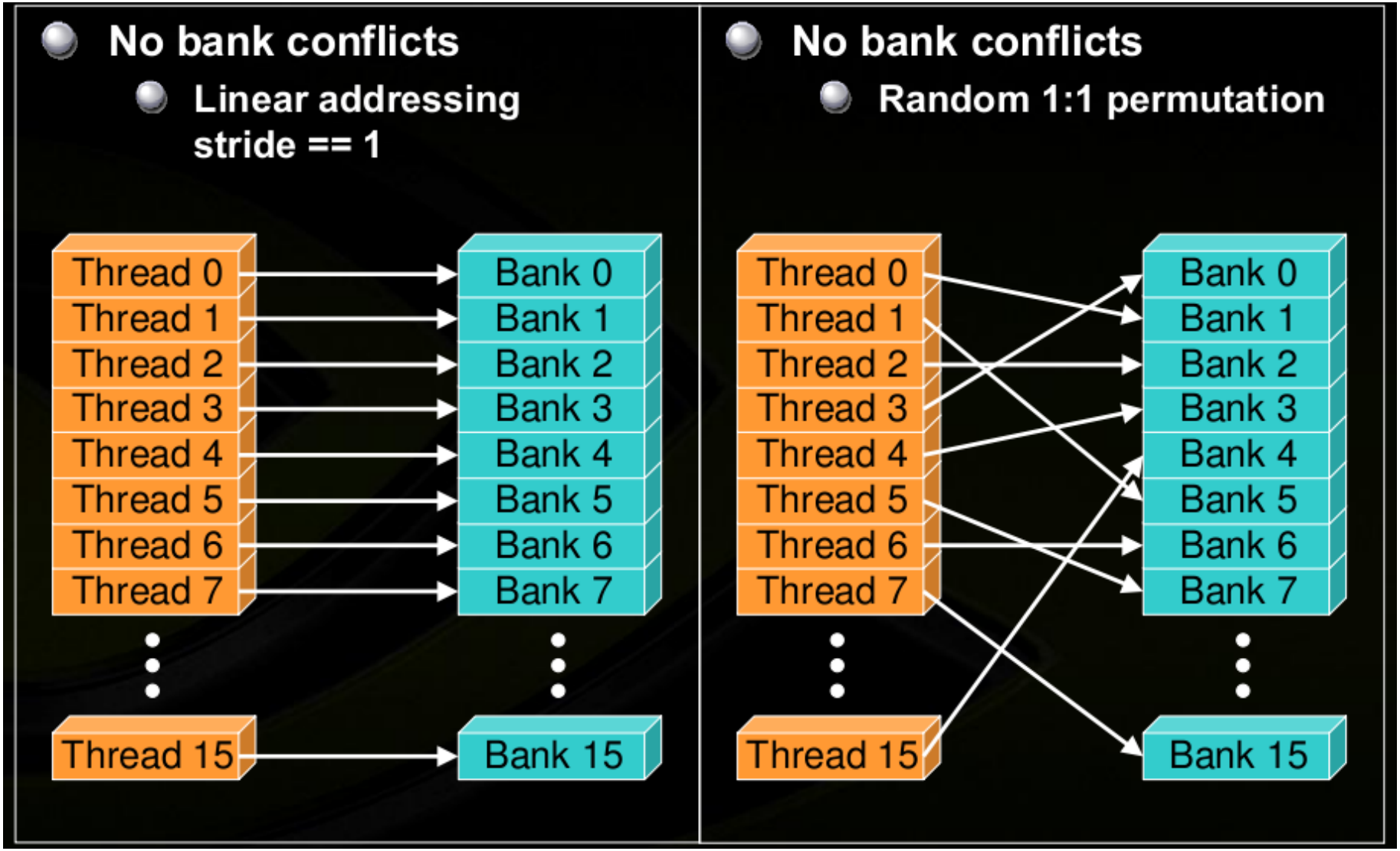
1. Shared memory is divided into 16 equally-sized memory modules, called **banks**, which are organized such that successive 32-bit words are assigned to successive banks.

2. These banks can be accessed simultaneously, and to achieve maximum bandwidth to and from shared memory the **threads in a half warp should access shared memory associated with different banks.**

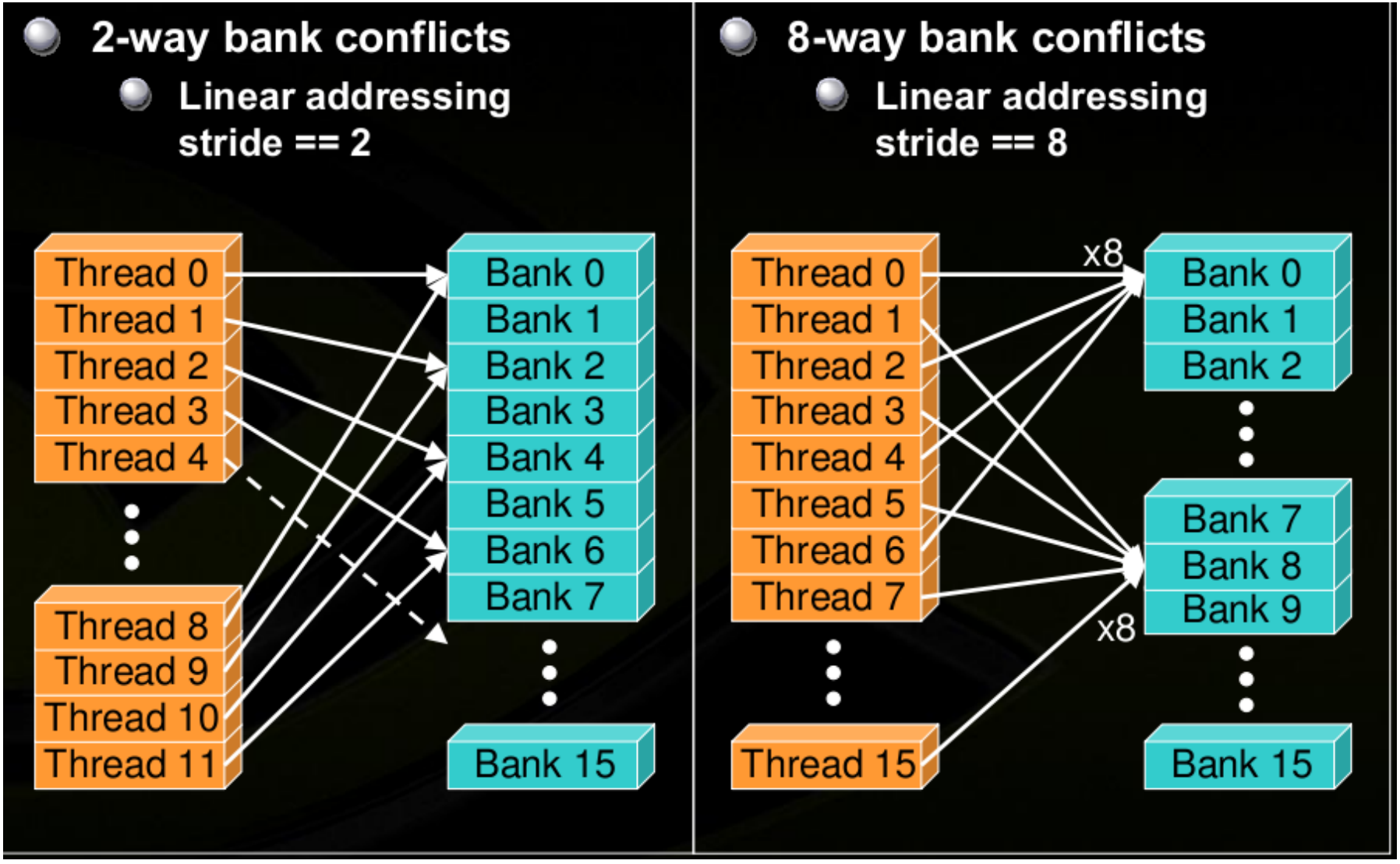
3. The **exception to this rule is** when all threads in a half warp read the same shared memory address, which results in a broadcast where the data at that address is sent to all threads of the half warp in one transaction.

4. One can use the `warp_serialize` flag when profiling CUDA applications to determine whether shared memory bank conflicts occur in any kernel.

Shared memory bank conflicts (2/6)



Shared memory bank conflicts (3/6)



Shared memory bank conflicts (4/6)

1. The coalesced transpose uses a 32×32 shared memory array of floats.
2. For this sized array, all data in columns `k` and `k+16` are mapped to the same bank.
3. As a result, when writing partial columns from `tile` in shared memory to rows in `odata` the half warp experiences a 16-way bank conflict and serializes the request.
4. A simple way to avoid this conflict is to pad the shared memory array by one column:

```
__shared__ float tile[TILE_DIM][TILE_DIM+1];
```

Shared memory bank conflicts (5/6)

- The padding does not affect shared memory bank access pattern when writing a half warp to shared memory, which remains conflict free,
- but by adding a single column now the access of a half warp of data in a column is also conflict free.
- The performance of the kernel, now coalesced and memory bank conflict free, is added to our table on the next slide.

Shared memory bank conflicts (6/6)

```
Device : Tesla M2050
Matrix size: 1024 1024, Block size: 32 8, Tile size: 32 32
                  Routine         Bandwidth (GB/s)
                     copy                  105.14
        shared memory copy                  104.49
           naive transpose                   18.82
        coalesced transpose                  51.42
  conflict-free transpose                    99.83
```

- While padding the shared memory array did eliminate shared memory bank conflicts, as was confirmed by checking the `warp_serialize` flag with the CUDA profiler, it has little effect (when implemented at this stage) on performance.
- As a result, there is still a large performance gap between the coalesced and shared memory bank conflict free transpose and the shared memory copy.

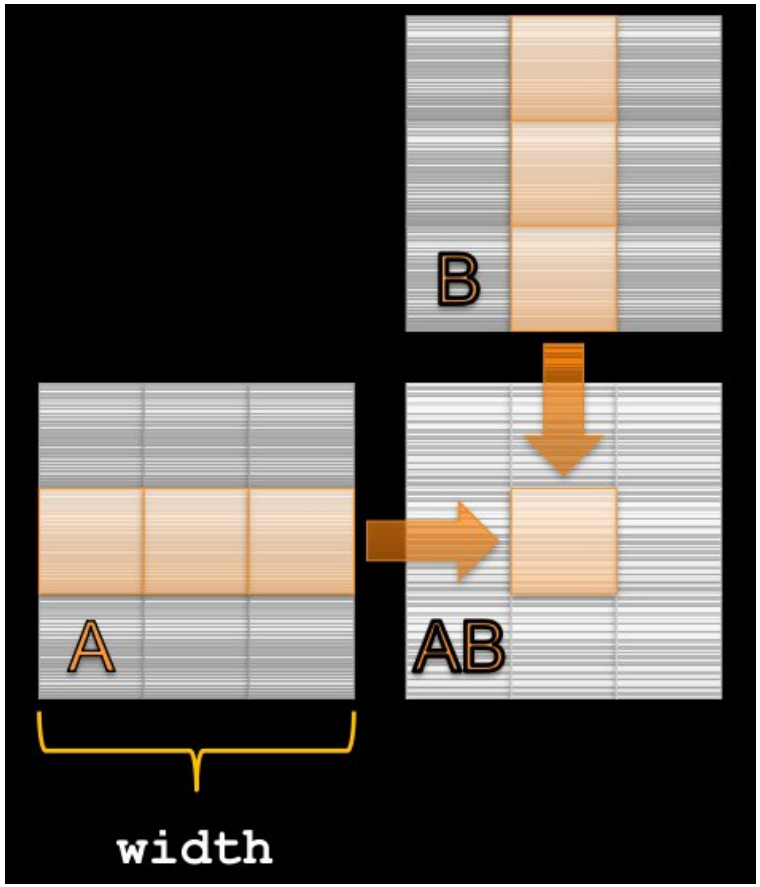
Plan

Matrix multiplication (1/16)

- The goals of this example are:
  - Understanding how to write a kernel for a non-toy example
  - Understanding how to map work (and data) to the thread blocks
  - Understanding the importance of using shared memory
- We start by writing a naive kernel for matrix multiplication which does not use shared memory.
- Then we analyze the performance of this kernel and realize that it is limited by the global memory latency.
- Finally, we present a more efficient kernel, which takes advantage of a tile decomposition and makes use of shared memory.

Matrix multiplication (2/16)

- Consider multiplying two rectangular matrices A and B with respective formats $m \times n$ and $n \times p$. Define $C = A \times B$.
- Principle: each thread computes an element of C through a 2D grid with 2D thread blocks.



Matrix multiplication (3/16)

```
__global__ void mat_mul(float *a, float *b,
                        float *ab, int width)
{
  // calculate the row & col index of the element
  int row = blockIdx.y*blockDim.y + threadIdx.y;
  int col = blockIdx.x*blockDim.x + threadIdx.x;
  float result = 0;
  // do dot product between row of a and col of b
  for(int k = 0; k < width; ++k)
    result += a[row*width+k] * b[k*width+col];
  ab[row*width+col] = result;
}
```

Matrix multiplication (4/16)

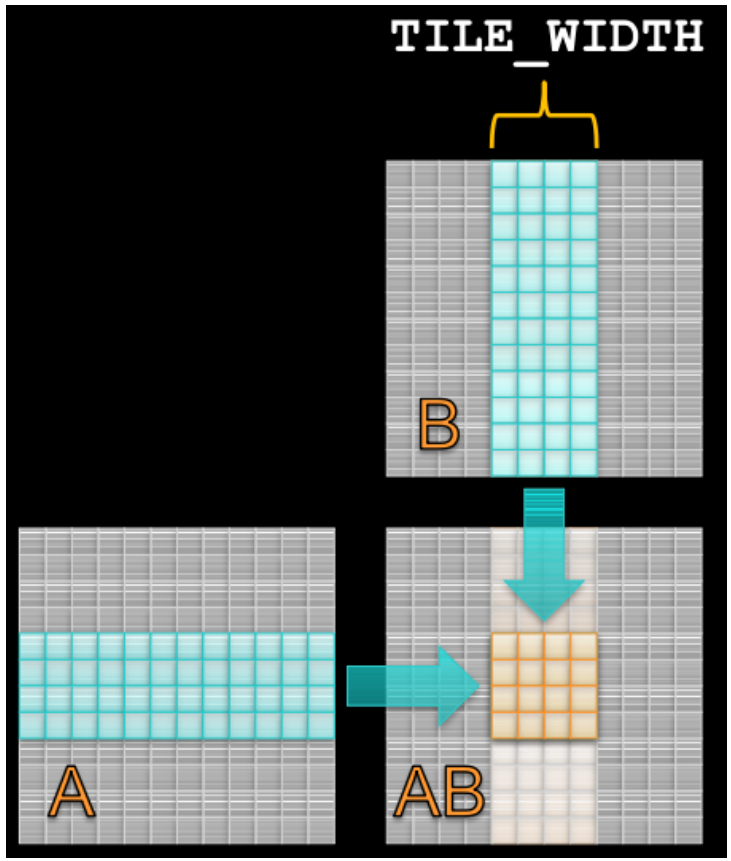
- Analyze the previous CUDA kernel for multiplying two rectangular matrices A and B with respective formats $m \times n$ and $n \times p$. Define $C = A \times B$.
- Each element of C is computed by one thread:
  - then each row of A is read p times and
  - each column of B is read m times, thus
  - $2\,m\,n\,p$ **reads in total for** $2\,m\,n\,p$ **flops.**
- Let t be an integer dividing m and p. We decompose C into $t \times t$ tiles. If tiles are computed one after another, then:
  - $(m/t)(t\,n)(p/t)$ slots are read in A
  - $(p/t)(t\,n)(m/t)$ slots are read in B, thus
  - $2m\,n\,p/t$ **reads in total for** $2\,m\,n\,p$ **flops.**
- For a CUDA implementation, $t = 16$ such that each tile is computed by one thread block.

Matrix multiplication (5/16)

- The previous explanation can be adapted to a particular GPU architecture, so as to estimate the performance of the first (naive) kernel.
- The first kernel has a **global memory access to flop ratio** (GMAC) of 8 Bytes / 2 ops, that is, 4 B/op.
- Suppose using a GeForce GTX 260, which has 805 GFLOPS peak performance.
- In order to reach **peak fp performance** we would need a memory bandwidth of $\mathrm{GMAC} \times \mathrm{Peak\ FLOPS} = 3.2$ TB/s.
- Unfortunately, we only have 112 GB/s of actual **memory bandwidth** (BW) on a GeForce GTX 260.
- Therefore an upper bound on the performance of our implementation is BW / GMAC $= 28$ GFLOPS.

Matrix multiplication (6/16)

- The picture below illustrates our second kernel
- Each thread block computes a tile in C, which is obtained as a dot product of tile-vector of A by a tile-vector of B.
- Tile size is chosen in order to maximize data locality.



Matrix multiplication (7/16)

- So a thread block computes a $t \times t$ tile of C.
- Each element in that tile is a dot-product of a row from A and a column from B.
- We view each of these dot-products as a sum of small dot products:

$$c_{i,j} = \sum_{k=o}^{t-1} a_{i,k} b_{k,j} + \sum_{k=t}^{2t-1} a_{i,k} b_{k,j} + \cdots \sum_{k=n-1-t}^{n-1} a_{i,k} b_{k,j}$$

- Therefore we fix ℓ and then compute $\sum_{k=\ell t}^{(\ell+1)t-1} a_{i,k} b_{k,j}$ for all i, j in the working thread block.
- We do this for $\ell = 0, 1, \ldots, (n/t - 1)$.
- This allows us to store the working tiles of A and B in shared memory.

Matrix multiplication (8/16)

- We assume that A, B, C are stored in row-major layout.
- Observe that for computing a tile in C our kernel code does need to know the number of rows in A.
- It just needs to know the `width` (number of columns) of A and B.

```
#define BLOCK_SIZE 16

     template <typename T>
__global__ void matrix_mul_ker(T* C, const T *A, const T *B,
        size_t wa, size_t wb)

    // Block index; WARNING: should be at most 2^16 - 1
    int bx = blockIdx.x;  int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x; int ty = threadIdx.y;
```

Matrix multiplication (9/16)

- We need the position in `*A` of the first element of the first working tile from A; we call it `aBegin`.
- We will need also the position in `*A` of the last element of the first working tile from A; we call it `aEnd`.
- Moreover, we will need the offset between two consecutive working tiles of A; we call it `aStep`.

```
int aBegin = wa * BLOCK_SIZE * by;

int aEnd = aBegin + wa - 1;

int aStep = BLOCK_SIZE;
```

Matrix multiplication (10/16)

- Similarly for B we have `bBegin` and `bStep`.
- We will not need a `bEnd` since once we are done with a row of A, we are also done with a column of B.
- Finally, we initialize the accumulator of the working thread; we call it `Csub`.

```
int bBegin = BLOCK_SIZE * bx;

int bStep = BLOCK_SIZE * wb;

int Csub = 0;
```

Matrix multiplication (11/16)

- The main loop starts by copying the working tiles of A and B to shared memory.

```
for(int a = aBegin, b = bBegin;  a <= aEnd; a += aStep, b += bStep)
    // shared memory for the tile of A
    __shared__ int As[BLOCK_SIZE][BLOCK_SIZE];

    // shared memory for the tile of B
    __shared__ int Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Load the tiles from global memory to shared memory
    // each thread loads one element of each tile
    As[ty][tx] = A[a + wa * ty + tx];
    Bs[ty][tx] = B[b + wb * ty + tx];

    // synchronize to make sure the matrices are loaded
    __syncthreads();
```

Matrix multiplication (12/16)

- Compute a small "dot-product" for each element in the working tile of C.

```
    // Multiply the two tiles together
    // each thread computes one element of the tile of C
    for(int k = 0; k < BLOCK_SIZE; ++k) {
        Csub += As[ty][k] * Bs[k][tx];
    }
    // synchronize to make sure that the preceding computatio
    // done before loading two new tiles of A dnd B in the ne
    __syncthreads();
}
```

Matrix multiplication (13/16)

- Once computed, the working tile of C is written to global memory.

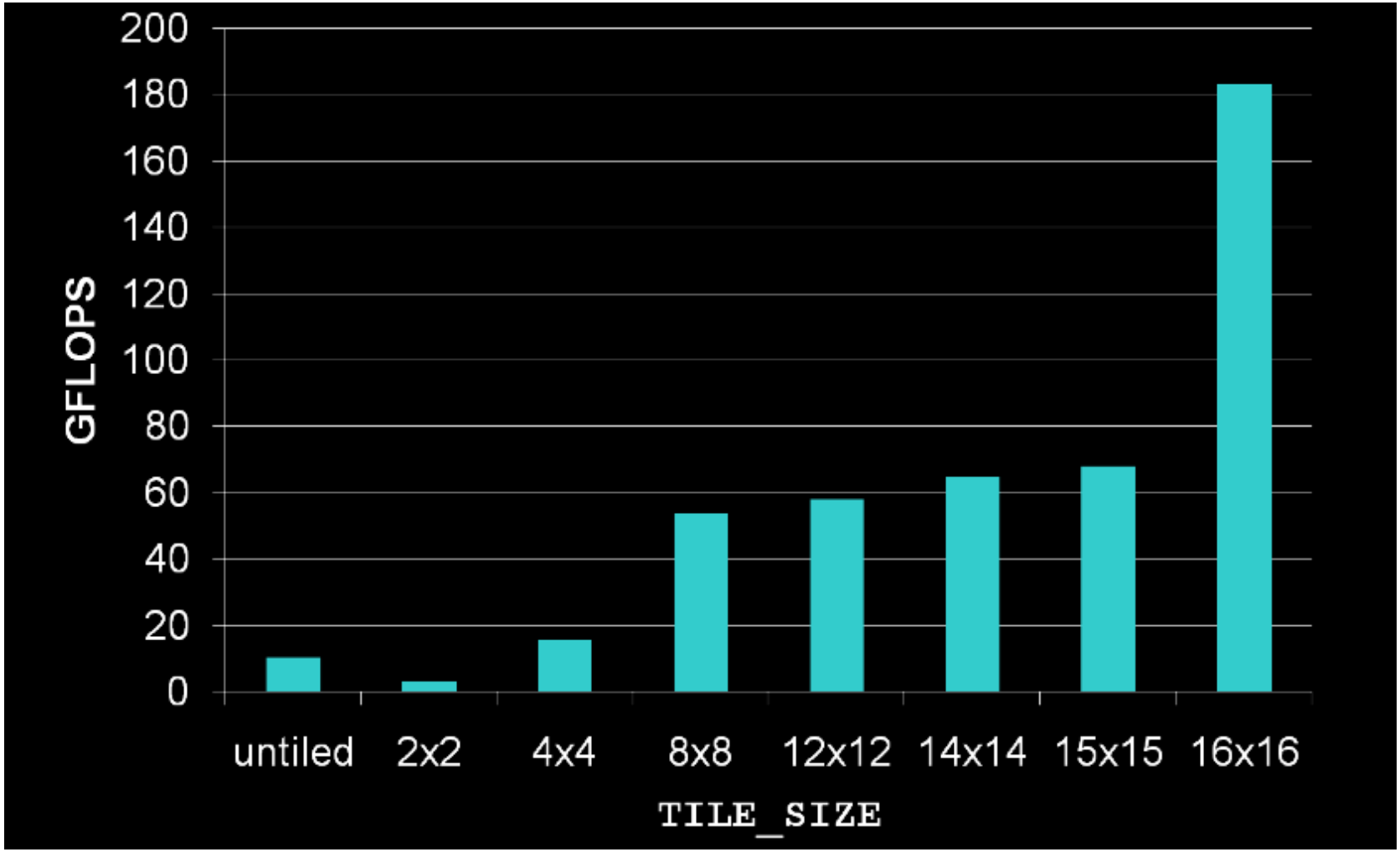
```
// Write the working tile of $C$ to global memory;
// each thread writes one element
int c = wb * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wb * ty + tx] = Csub;
```

Matrix multiplication (14/16)

- Each thread block should have many threads:
  - $\mathrm{TILE\_WIDTH} = 16$ implies $16 \times 16 = 256$ threads
- There should be many thread blocks:
  - A 1024×1024 matrix would require 4096 thread blocks.
  - Since one streaming multiprocessor (SM) can handle 768 threads, each SM will process 3 thread blocks, leading it **full occupancy**.
- Each thread block performs 2×256 reads of a 4-byte float while performing $256 \times (2 \times 16) \quad = \quad 8,192$ fp ops:
  - Memory bandwidth is no longer limiting factor

Matrix multiplication (15/16)

- Experimentation performed on a GT200.
- **Tiling** and using **shared memory** were clearly worth the effort.



Matrix multiplication (16/16)

- Effective use of different memory resources reduces the number of accesses to global memory
- But these resources are finite!
- The more memory locations each thread requires, the fewer threads an SM can accommodate.

Resource	Per GT200 SM	Full Occupancy on GT200
Registers	16384	<= 16384 / 768 threads = 21 per thread
`__shared__` Memory	16KB	<= 16KB / 8 blocks = 2KB per block

Plan

Prefix sum

Prefix sum of a vector: specification

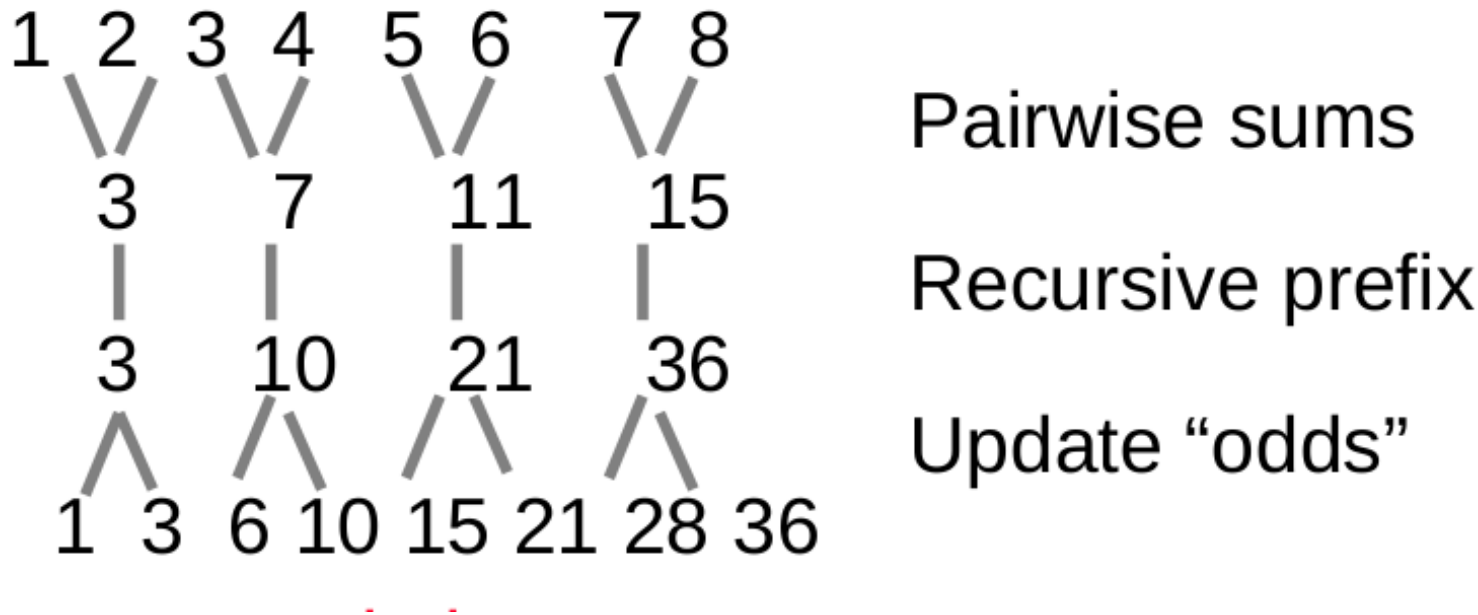
Input: a vector $\vec{x} = (x_1, x_2, \ldots, x_n)$

Ouput: the vector $\vec{y} = (y_1, y_2, \ldots, y_n)$ such that $y_i = \sum_{i=1}^{j=i} x_j$ for $1 \leq j \leq n$.

Prefix sum of a vector: example

The prefix sum of $\vec{x} = (1, 2, 3, 4, 5, 6, 7, 8)$ is $\vec{y} = (1, 3, 6, 10, 15, 21, 28, 36)$.

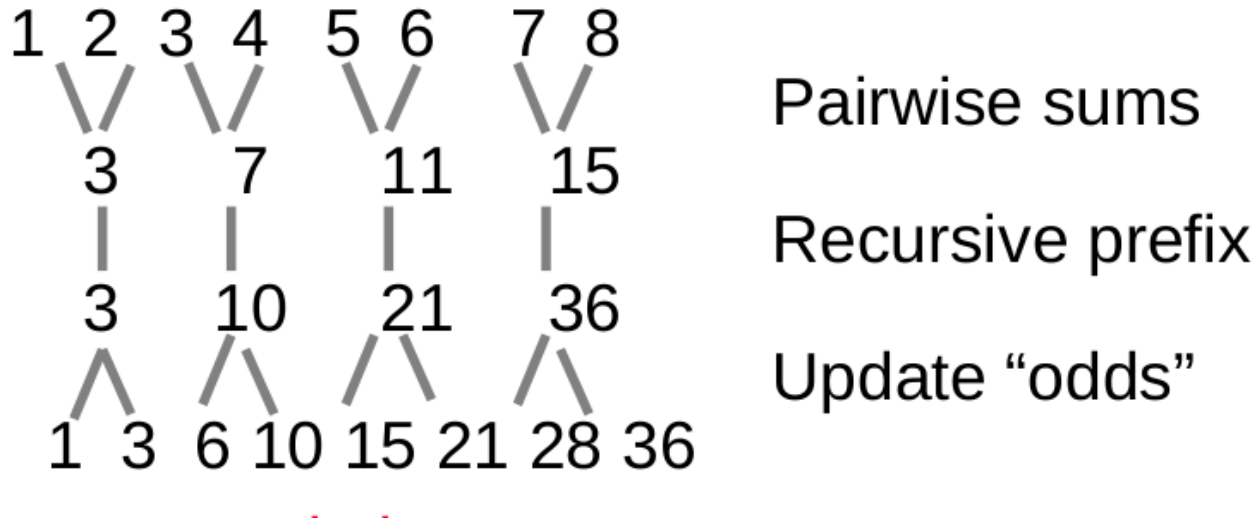
Prefix sum: a recursive work-efficient algorithm (1/2)



Algorithm

- Input: $x[1], x[2], \ldots, x[n]$ where n is a power of 2.
- Step 1: $x[k] = x[k] + x[k-1]$ for all even k's.
- Step 2: Recursive call on $x[2], x[4], \ldots, x[n]$
- Step 3: $x[k-1] = x[k] - x[k-1]$ for all even k's.

Prefix sum: a recursive work-efficient algorithm (2/2)



Analysis

- Since the recursive call is applied to an array of size $n/2$, the total number of recursive calls is $\log(n)$.
- Before the recursive call, one performs $n/2$ additions
- After the recursive call, one performs $n/2$ subtractions
- Elementary calculations show that this recursive algorithm performs at most a total of $2n$ additions and subtractions
- Thus, this algorithm is work-efficient. In addition, it can run in $2\log(n)$ parallel steps.

Application to parallel addition (1/2)

Example						Notation			
1	0	1	1	1	Carry	c_2	c_1	c_0	
	1	0	1	1	1 First Int	a_3	a_2	a_1	a_0
	1	0	1	0	1 Second Int	b_3	b_2	b_1	b_0

Application to parallel addition (2/2)

Example						Notation			
1	0	1	1	1	Carry	c_2	c_1	c_0	
	1	0	1	1	1 First Int	a_3	a_2	a_1	a_0
	1	0	1	0	1 Second Int	a_3	b_2	b_1	b_0

$c_{-1} = 0$

(addition mod 2)

for i = 0 : n-1

$s_i = a_i + b_i + c_{i-1}$

$c_i = a_i b_i + c_{i-1}(a_i + b_i)$

end

$$\begin{bmatrix} c_i \\ 1 \end{bmatrix} = \begin{bmatrix} a_i + b_i & a_i b_i \\ 0 & 1 \end{bmatrix} \begin{bmatrix} c_{i-1} \\ 1 \end{bmatrix}$$

Call M_i the above matrix. Computing all $M_i \cdots M_2 M_1$ computes all carries in $\log(n)$ steps by means of parallel prefix sum.

Parallel addition of big integers: experimental results

- Number of words per big integer: 256
- Number of pairs to be added: 1024
- test cases: random and 4 corner cases, each one run 100 times
- Architecture: NVIDIA Tesla 2050C
- Implementation bases on 32-bit:
  - Average GPU time = 0.166726ms
  - Average CPU time = 1.770852ms
- Implementation bases on 64-bit:
  - Average GPU time = 0.330151ms
  - Average CPU time = 1.944842ms

Plan

Optimize algorithms targeting GPU-like many-core devices

Desirable goals

- Given a CUDA code, an experimented programmer may attempt well-known strategies to improve the code performance in terms of arithmetic intensity and memory bandwidth.
- Given a CUDA-like algorithm, one would like to derive code for which much of this optimization process has been lifted at the design level, i.e. before the code is written.

Problem

We need a model of computation which

- captures the computer hardware characteristics that have a dominant impact on program performance.
- combines its complexity measures (work, span) so as to determine the *best* algorithm among different possible algorithmic solutions to a given problem.

Challenges in designing a model of computation for GPUs

Theoretical aspects

- GPU-like architectures introduces many machine parameters (like memory sizes, number of cores), and too many could lead to intractable calculations.
- GPU-like code depends also on program parameters (like number of threads per thread-block) which specify how the work is divided among the computing resources.

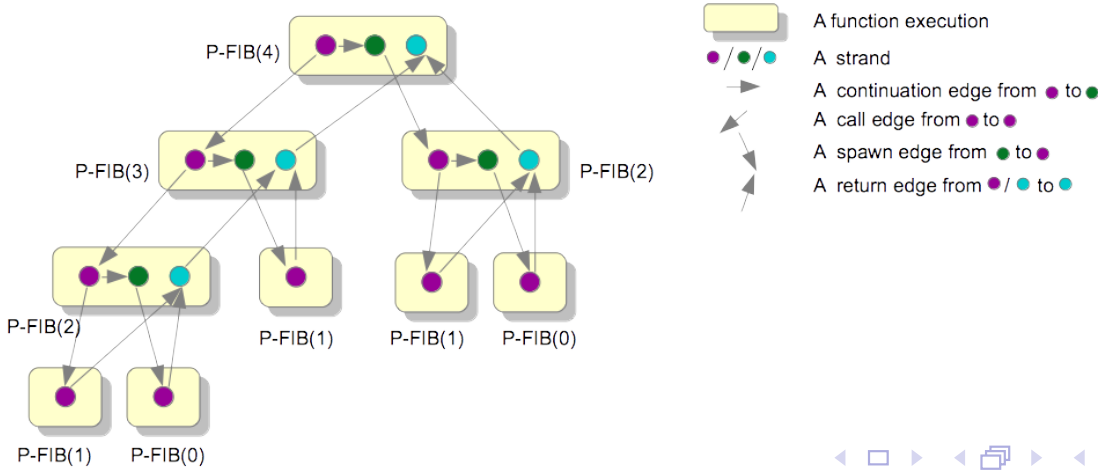
Practical aspects

- One wants to avoid answers like: *Algorithm 1 is better than Algorithm 2* providing that the machine parameters satisfy a system of constraints.
- We prefer analysis results independent of machine parameters.
- We expect that this should be achieved by selecting program parameters in appropriate ranges.

Fork-join model

This model has become popular with the development of the concurrency platform CilkPlus, targeting multi-core architectures.

- The *work* T_1 is the total time to execute the entire program on one processor.
- The *span* T_∞ is the longest time to execute along any path in the DAG.
- We recall that the Graham-Brent theorem states that the running time T_P on P processors satisfies $T_\mathsf{P} \leq T_1/\mathsf{P} + T_\infty$. A refinement of this theorem captures scheduling and synchronization costs, that is, $T_\mathsf{P} \leq T_1/\mathsf{P} + 2\delta\widehat{T_\infty}$, where δ is a constant and $\widehat{T_\infty}$ is the burdened span.



Parallel random access machine (PRAM) model

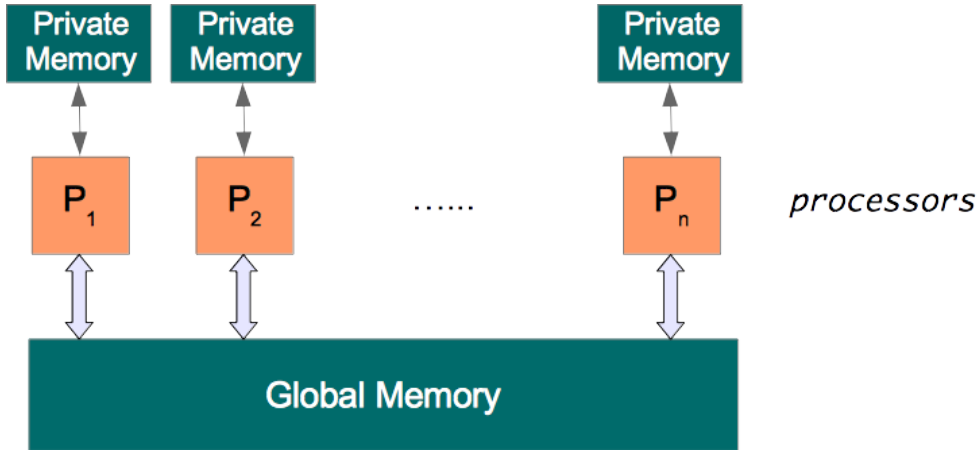


Figure: Abstract machine of PRAM model

- Instructions on a processor execute in a 3-phase cycle: read-compute-write.
- Processors access to the global memory in a unit time (unless an access conflict occurs).
- These strategies deal with read/write conflicts to the same global memory cell: EREW, CREW and CRCW (exclusive or concurrent).
- A refinement of PRAM integrates communication delay into the computation time.

Recent many-core machine models

Hong and Kim 2009 present an analytical model to estimate the execution time of parallel programs on GPU architectures.

- Their estimated running time is based on the estimated CPI (cycles per instruction).
- It also requires machine parameters, such as the specifications of a GPU card.

Ma, Agrawal and Chamberlain 2014 introduce the threaded many-core memory (TMM) model which retains many important characteristics of GPU-type architectures.

- In TMM analysis, the running time of an algorithm is estimated by choosing the maximum quantity among the work, span and amount of memory accesses. No Graham-Brent theorem-like is provided.
- Such running time estimates depend on the machine parameters.

Plan

A many-core machine (MCM) model

We propose a many-core machine (MCM) model which aims at

- tuning program parameters to minimize parallelism overheads of algorithms targeting GPU-like architectures as well as
- comparing different algorithms independently of the value of machine parameters of the targeted hardware device.

In the design of this model, we insist on the following features:

- Two-level DAG programs
- Parallelism overhead
- A Graham-Brent theorem

(Sardar Anisul Haque, MMM, Ning Xie; ParCo 2015)

Plan

Characteristics of the abstract many-core machines

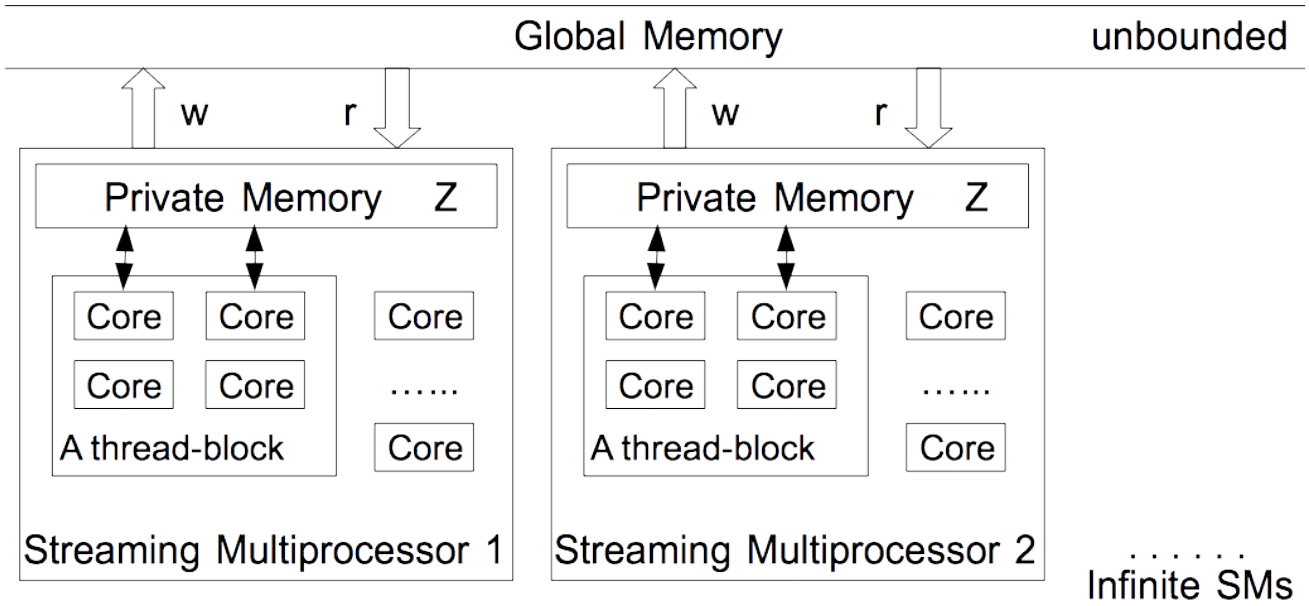


Figure: A many-core machine

- It has a global memory with high latency and low throughput while private memories have low latency and high throughput

Characteristics of the abstract many-core machines

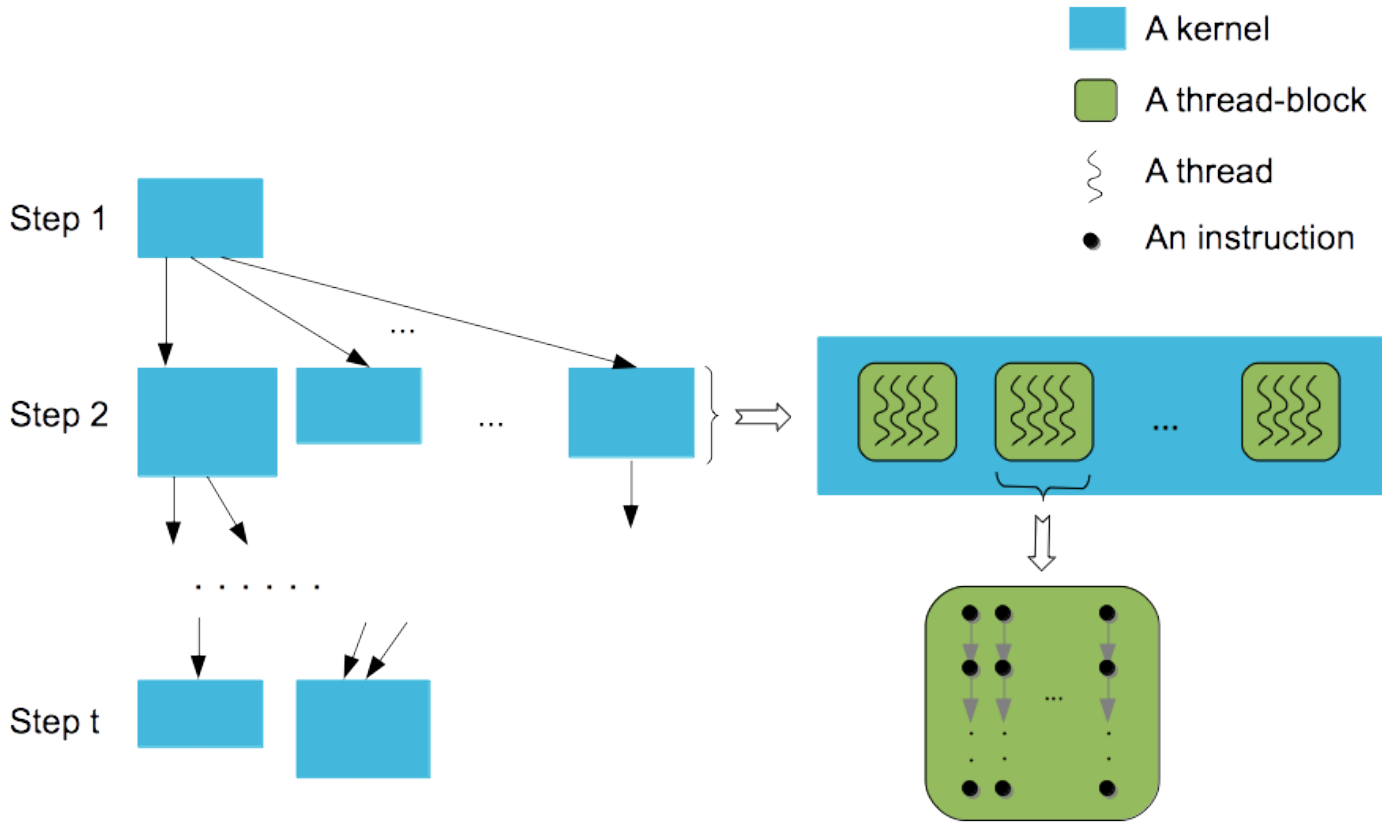


Figure: Overview of a many-core machine program, also called *kernel DAG*

Characteristics of the abstract many-core machines

Synchronization costs

- It follows that MCM kernel code needs no synchronization statement.
- Consequently, the only form of synchronization taking place among the threads executing a given thread-block is implied by code divergence.
- An MCM machine handles code divergence by eliminating the corresponding conditional branches via code replication, and the corresponding cost will be captured by the complexity measures (work, span and parallelism overhead) of the MCM model.

Characteristics of the abstract many-core machines

Scheduling costs

- The kernel DAG defining an MCM program $\mathcal{P}$ is assumed to be known when $\mathcal{P}$ starts to execute.
- Scheduling $\mathcal{P}$'s kernels onto the SMs can be done in time $O(\Gamma)$ where Γ is the total length of $\mathcal{P}$'s kernel code.
- We neglect those costs.

Machine parameters of the abstract many-core machines

Z: Private memory size of any SM

- It sets up an upper bound on several program parameters, for instance, the number of threads of a thread-block or the number of words in a data transfer between the global memory and the private memory of a thread-block.

U: Data transfer time

- Time (expressed in clock cycles) to transfer one machine word between the global memory and the private memory of any SM, that is, $U > 0$.
- As an abstract machine, the MCM aims at capturing either the best or the worst scenario for data transfer time of a thread-block, that is,

$$\begin{array}{rcl} T_D & \leq & (\alpha + \beta)\, U, \text{ if coalesced accesses occur;} \\ & \text{or} & \ell\,(\alpha + \beta)\, U, \text{ otherwise,} \end{array}$$

Plan

Complexity measures for the many-core machine model

For any kernel $\mathcal{K}$ of an MCM program,

- **work** $W(\mathcal{K})$ is the total number of local operations of all its threads;
- **span** $S(\mathcal{K})$ is the maximum number of local operations of one thread;
- **parallelism overhead** $O(\mathcal{K})$ is the total data transfer time among all its thread-blocks.

For the entire program $\mathcal{P}$,

- **work** $W(\mathcal{P})$ is the total work of all its kernels;
- **span** $S(\mathcal{P})$ is the longest path, counting the weight (span) of each vertex (kernel), in the kernel DAG;
- **parallelism overhead** $O(\mathcal{P})$ is the total parallelism overhead of all its kernels.

Characteristic quantities of the thread-block DAG

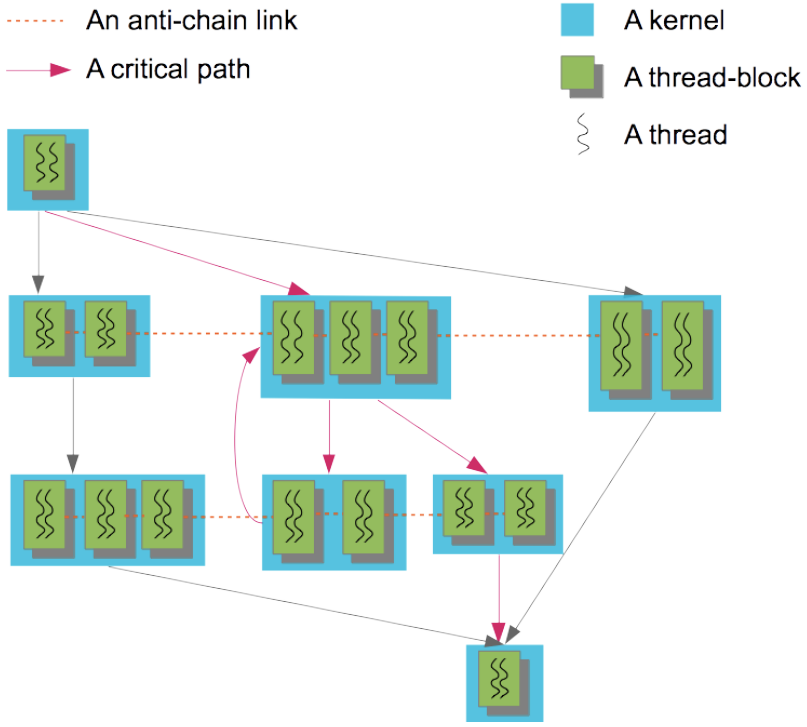


Figure: Thread-block DAG of a many-core machine program

$\mathrm{N}(\mathcal{P})$: number of vertices in the thread-block DAG of $\mathcal{P}$,

$\mathrm{L}(\mathcal{P})$: critical path length (where length of a path is the number of edges in that path) in the thread-block DAG of $\mathcal{P}$.

Complexity measures for the many-core machine model

Theorem (A Graham-Brent theorem with parallelism overhead)

We have the following estimate for the running time T_{P} *of the program* $\mathcal{P}$ *when executed on* P *SMs:*

$$T_{\mathsf{P}} \leq (\mathrm{N}(\mathcal{P})/\mathsf{P} + \mathrm{L}(\mathcal{P}))\mathrm{C}(\mathcal{P}) \tag{1}$$

where $\mathrm{C}(\mathcal{P})$ *is the maximum running time of local operations (including read/write requests) and data transfer by one thread-block.*

Corollary

Let K *be the maximum number of thread-blocks along an anti-chain of the thread-block DAG of* $\mathcal{P}$*. Then the running time* $T_{\mathcal{P}}$ *of the program* $\mathcal{P}$ *satisfies:*

$$T_{\mathcal{P}} \leq (\mathrm{N}(\mathcal{P})/K + \mathrm{L}(\mathcal{P}))\mathrm{C}(\mathcal{P}) \tag{2}$$

Plan

Tuning a program parameter with the MCM model

For an MCM program $\mathcal{P}$ depending on a program parameter s varying in a range $\mathcal{S}$.

- Let s_0 be an "initial" value of s corresponding to an instance $\mathcal{P}_0$ of $\mathcal{P}$.
- Assume the work ratio W_{s_0}/W_s remains essentially constant meanwhile the parallelism overhead O_s varies more substantially, say $O_{s_0}/O_s \in \Theta(s - s_0)$.
- Then, we determine a value $s_{\min} \in \mathcal{S}$ maximizing the ratio O_{s_0}/O_s.
- Next, we use our version of Graham-Brent theorem to confirm that the upper bound for the running time of $\mathcal{P}(s_{\min})$ is less than that of $\mathcal{P}(s_0)$.

Plan

Sequential algorithm

We denote by a and b two univariate polynomials over a (finte) field, with sizes $n \geq m$:

$$a = a_1 X^{n-1} + \cdots + a_1 X + a_n \quad \text{and} \quad b = b_1 X^{m-1} + \cdots + b_1 X + b_m. \quad (3)$$

We compute their product $f = a \times b$.

			$a =$		X^5+	$8X^4+$	$2X^3+$	$2X^2+$	$6X+$	7
			$b =$		X^5+	$2X^4+$	$4X^3+$	X^2+	$3X+$	2
					$2X^5+$	$16X^4+$	$4X^3+$	$4X^2+$	$12X+$	14
				$3X^6+$	$24X^5+$	$6X^4+$	$6X^3+$	$18X^2+$	$21X$	
			X^7+	$8X^6+$	$2X^5+$	$2X^4+$	$6X^3+$	$7X^2$		
		$4X^8+$	$32X^7+$	$8X^6+$	$8X^5+$	$24X^4+$	$28X^3$			
	$2X^9+$	$16X^8+$	$4X^7+$	$4X^6+$	$12X^5+$	$14X^4$				
$X^{10}+$	$8X^9+$	$2X^8+$	$2X^7+$	$6X^6+$	$7X^5$					
$X^{10}+$	$10X^9+$	$22X^8+$	$39X^7+$	$29X^6+$	$55X^5+$	$62X^4+$	$44X^3+$	$29X^2+$	$33X+$	14

Table: A plain multiplication $n = m = 6$.

Principle of parallelization

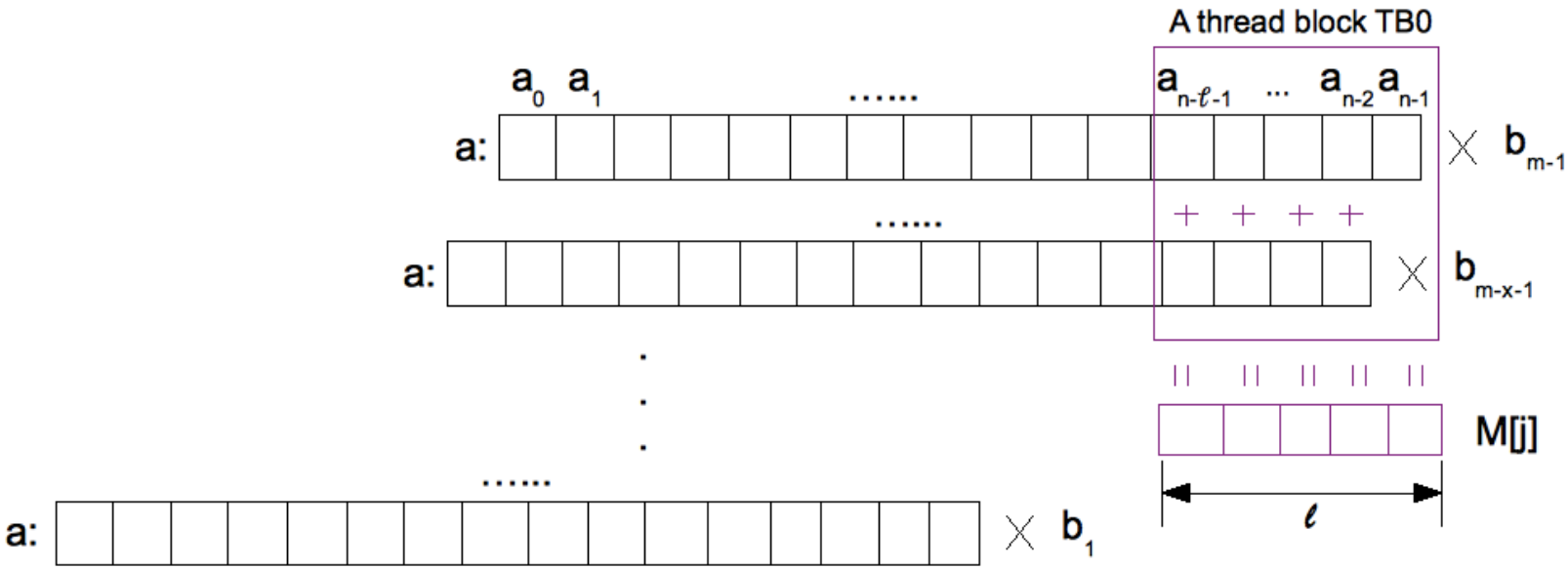


Figure: Dividing the work among threadblocks and threads.

- *Multiplication phase*: every coefficient of a is multiplied with every coefficients of b; each thread accumulates s partial sums into an auxiliary array M.
- *Addition phase*: these partial sums are added together repeatedly to form the polynomial f.

Complexity analysis

The work, span and parallelism overhead ratios between $s_0 = 1$ (initial program) and an arbitrary s are, respectively[1],

$$\frac{\mathrm{W}_1}{\mathrm{W}_s} = \frac{n}{n+s-1},$$

$$\frac{\mathrm{S}_1}{\mathrm{S}_s} = \frac{\log_2(m)+1}{s\left(\log_2\left(m/s\right)+2\,s-1\right)},$$

$$\frac{\mathrm{O}_1}{\mathrm{O}_s} = \frac{n\,s^2\left(7\,m-3\right)}{\left(n+s-1\right)\left(5\,m\,s+2\,m-3\,s^2\right)}.$$

- Let m esacpe to infinity with $m \leq n$.
- Increasing s leaves work essentially constant, while span increases and parallelism overhead decreases in the same order.
- Hence, should s be large or close to $s_0 = 1$?

[1] See the detailed analysis in the form of executable MAPLE worksheets of three applications: `http://www.csd.uwo.ca/~nxie6/projects/mcm/`

Narrowing the value of the program parameter (1/2)

Applying our version of the Graham-Brent theorem, the ratio R of the estimated running times on $\Theta(\frac{(n+s-1)\,m}{\ell\,s^2})$ SMs is

$$R = \frac{(m\,\log_2(m) + 3\,m - 1)\,(1 + 4\,U)}{(m\,\log_2(\frac{m}{s}) + 3\,m - s)\,(2\,U\,s + 2\,U + 2\,s^2 - s)}.$$

which is asymptotically equivalent to $\frac{2\,U\,\log_2(m)}{s\,(s+U)\,\log_2(m/s)}$.

- This latter ratio is less than 1 for $s > 1$, since $U > 0$.
- In other words, increasing s makes the algorithm performance worse.

Narrowing the value of the program parameter (2/2)

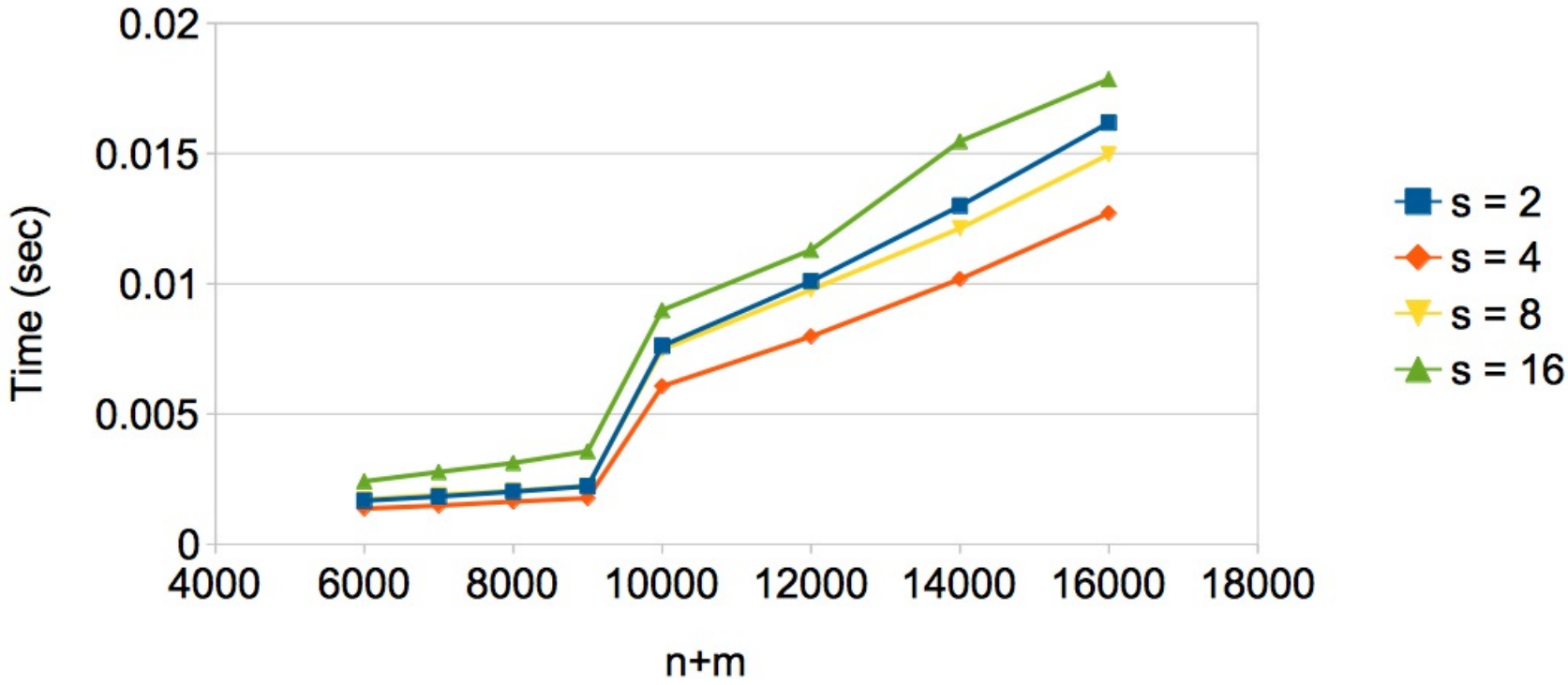


Figure: Running time of the plain polynomial multiplication algorithm with polynomials a ($\deg(a) = n - 1$) and b ($\deg(b) = m - 1$) and the parameter s on GeForce GTX 670.

Plan

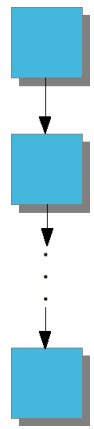
Plain division for polynomials

Given two polynomials a and b over a finite field $\mathbb{K}$, where $\deg(a) = n - z1$, and $\deg(b) = m - 1$, we compute the remainder in the Euclidean division of a by b, using:

- a naive division algorithm
- an optimized division algorithm

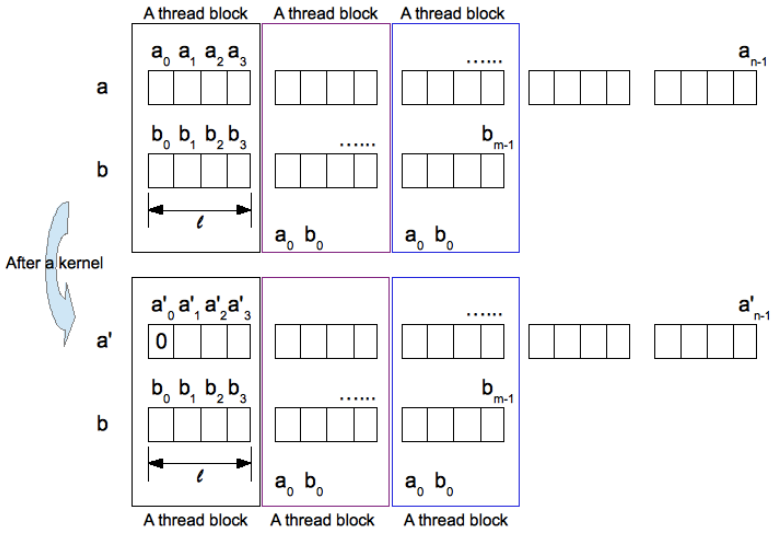
We assume that

- b is not zero
- $n \geq m$



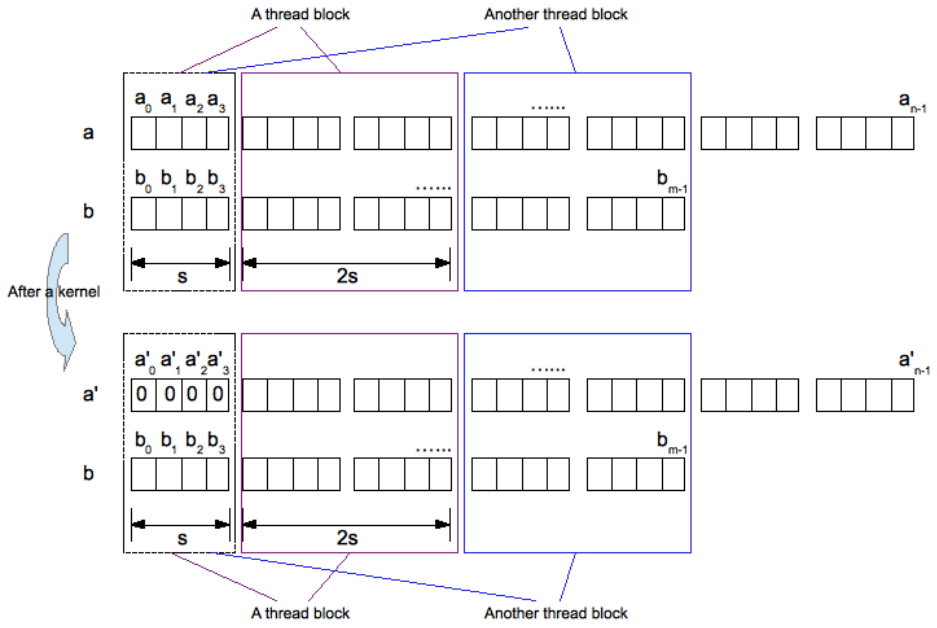
Naive and optimized approaches

Naive Division Algorithm



- Each kernel performs 1 division step
- $n - m + 1$ kernel calls are executed sequentially

Optimized Division Algorithm



- Each kernel performs (at least) s division steps
- $\lceil \frac{n-m+1}{s} \rceil$ kernel calls are executed sequentially

Complexity analysis

We obtain the work ratio and the overhead ratio as

$$\frac{W_{\mathrm{nai}}}{W_{\mathrm{opt}}} = \frac{8\,(Z+1)}{9\,Z+7} \quad \text{and} \quad \frac{O_{\mathrm{nai}}}{O_{\mathrm{opt}}} = \frac{20}{441}\,Z$$

Applying Theorem 1,

$$R = \frac{(N_{\mathrm{nai}}/p + L_{\mathrm{nai}}) \cdot C_{\mathrm{nai}}}{(N_{\mathrm{opt}}/p + L_{\mathrm{opt}}) \cdot C_{\mathrm{opt}}} = \frac{2}{3}\,\frac{(3+5\,U)\,(2\,m+Z\,p)\,Z}{(Z+21\,U)\,(7\,m+2\,Z\,p)}$$

When m escapes to infinity, the ratio R is equivalent to

$$\frac{4}{21}\,\frac{(3+5\,U)\,Z}{Z+21\,U}$$

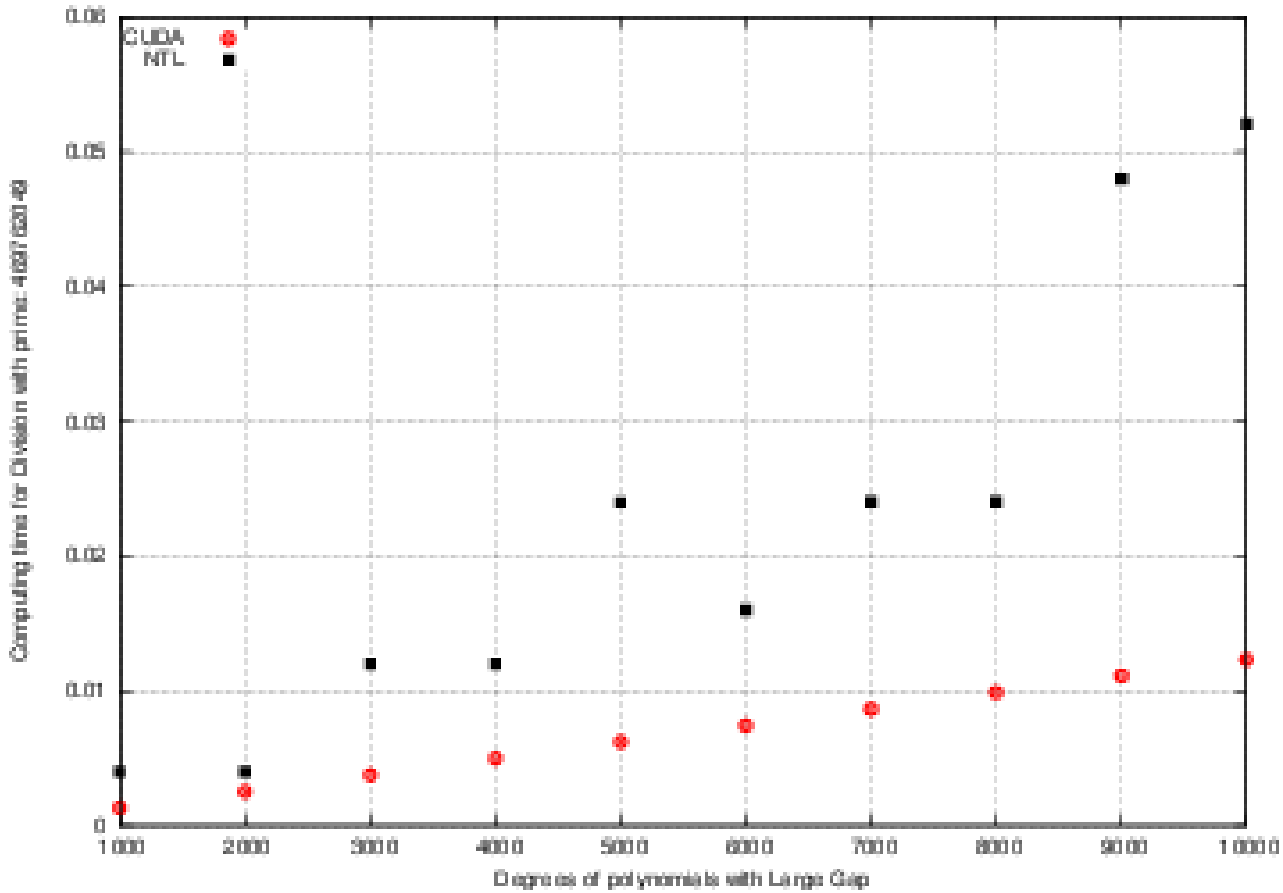
- We observe that this latter ratio is larger than 1 if and only if $Z > \frac{441\,U}{20\,U-9}$ holds
- The optimized algorithm is overall better than the naive one

Experimental results

Optimized vs naive

Optimized division is almost 4 times faster than naive division with $s = 256$.

Optimized vs NTL library



Plan

Principle of parallelization

Let $s > 0$. We proceed by repeatedly calling a subroutine which

- takes as input a pair (a, b) of polynomials and
- returns another pair (a', b') of polynomials such that $\gcd(a, b) = \gcd(a', b')$ and, either $b' = 0$ or we have $\deg(a') + \deg(b') \leq \deg(a) + \deg(b) - s$.
- When $s = \Theta(\ell)$ (the number of threads per thread-block), the work is increased by a constant factor and the parallelism overhead will reduce by a factor in $\Theta(s)$.
- Further, the estimated running time ratio T_1/T_s on $\Theta(\frac{m}{\ell})$ SMs is greater than 1 if and only if $s > 1$.

Analysis of the Euclidean algorithm

We obtain the work ratio and the overhead ratio, replacing m by n as

$$\frac{W_{\mathrm{nai}}}{W_{\mathrm{opt}}} = \frac{(284\,Z+2)\,n^2+(Z-2)\,n}{(1296\,Z+7488)\,n^2+(348\,Z^2+2208\,Z)\,n-(115\,Z^3+616\,Z^2)}$$

$$\frac{O_{\mathrm{nai}}}{O_{\mathrm{opt}}} = \frac{5}{48}\,\frac{Z(2\,n+2+Z)}{6\,n+Z}$$

- As n escapes to infinity, the additional work $W_{\mathrm{opt}} - W_{\mathrm{nai}}$ is only a portion of W_{nai},
- Meanwhile the data transfer overhead decreases as Z increases.

Applying Theorem 1, when n escapes to infinity, the ratio R is equivalent to

$$R = \frac{(N_{\mathrm{nai}}/p + L_{\mathrm{nai}}) \cdot C_{\mathrm{nai}}}{(N_{\mathrm{opt}}/p + L_{\mathrm{opt}}) \cdot C_{\mathrm{opt}}} \simeq \frac{(3+5\,U)\,Z}{9\,(Z+16\,U)}$$

Experimental results (1/2)

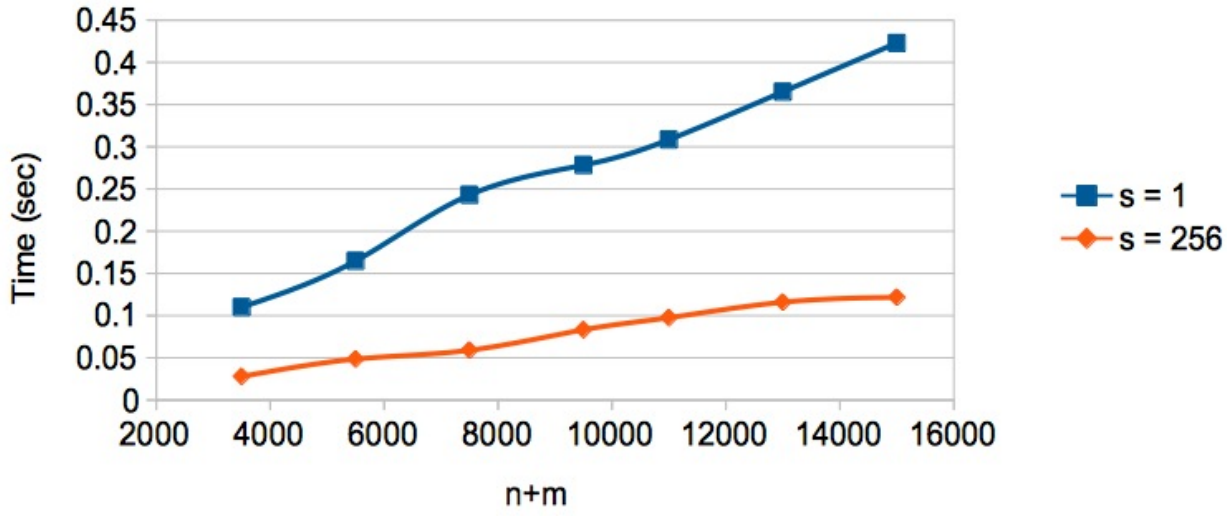


Figure: Running time on GeForce GTX 670 of our multithreaded Euclidean algorithm for univariate polynomials of sizes n and m over $\mathbb{Z}/p\mathbb{Z}$ where p is a 30-bit prime; the program parameter takes values $s = 1$ and $s = 256$.

Experimental results (2/2)

Optimized vs NTL library