CUDA

Parallel programming on pixels

Performance Tuning

- Machine Configuration
- Accurate Timing
- Block size
- Control flow
- Memory access
- Unroll
- Double buffering

Machine Configuration

- Use cudaGetDeviceCount(&device_count) to determine the number of devices in the system.
- Use cudaGetDeviceProperties(cudaDeviceProp *, int deviceid) to determine the important parameters in performance tuning.
  - Warp size
  - Maximum threads per block

cudaDeviceProp

- char name[256];
- size_t totalGlobalMem;
- size_t sharedMemPerBlock;
- int regsPerBlock;
- int warpSize;
- size_t memPitch;
- int maxThreadsPerBlock;
- int maxThreadsDim[3];
- int maxGridSize[3];

cudaDeviceProp

- size_t totalConstMem;
- int major;
- int minor;
- int clockRate;
- size_t textureAlignment;
- int deviceOverlap;
- int multiProcessorCount;

Acurate Timing

- Use cuda events for accurate timing, since event has time information.
- Place start event into the event stream 0.
- Place the code for timing.
- Place end event into the event stream 0.
- Wait for the end event to finish.
- Compute the elapse time between start and end.

Event Routines

- Declared as type cudaEvent_t
- Created using cudaEventCrteate(*cudaEvent_t)
- Record into the operation stream by cudaEventRecord(cudaEvent_t, int stream).
  - Use stream 0, which is for all events.
- Synchronized with event using cudaEventSynchronize(cudaEvent_t).
- Compute the elapsed time using cudaEventElapsedTime(float *elapsed_time, cudaEvent_t start, cudaEvent_t end).
- Destroyed with cudaEnentDestroy(cudaEvent_t).

An Example

```
float time;
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
```

An Example

```
cudaEventRecord(start, 0);
multiply <<< grids, blocks >>> ((int (*)[N])device_A, (int
    (*)[N])device_B, (int (*)[N])device_C);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time, start, stop);
printf("the multiplcaition takes %f seconds¥n", time);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Block Size

- How many threads should go into a block?
  - Limitation on the resource of multiprocessors

Thread Block

- Up to 512 threads per block
- Up to three dimensions
- Has shared memory
- Can synchronize with __syncthread()

Multiprocessor

- So called Streaming Multiprocessors (SM)
- A SM can run a limited number of blocks and threads.
  - 8 blocks and 768 threads for G80
- SM will maintain thread and block id for all thread blocks running on it, and schedule these threads for execution.

Warps

- Thread are scheduled in unit of 32 (called Warp).
- CUDA Programmer cannot see this, but should be aware of this.
- Only one warp (from all blocks assigned to a SM) will execute on a SM at any given time.

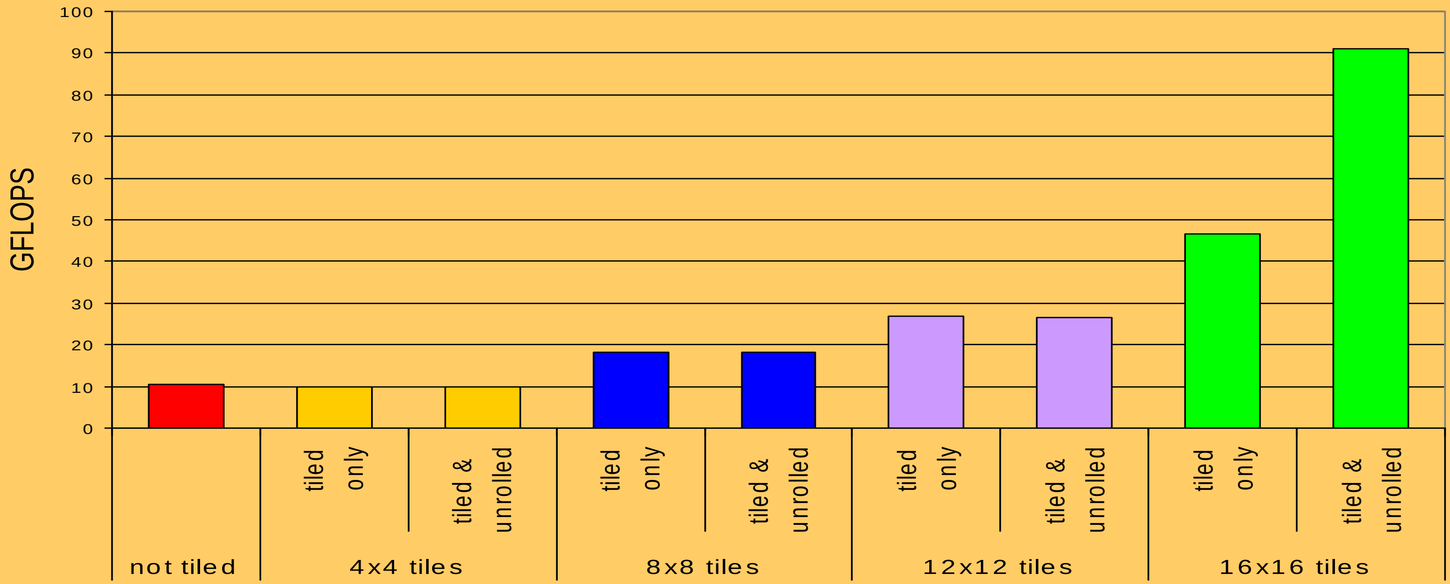
Warp Scheduling

- Operand-ready warps are eligible for execution.
- It takes 4 cycles to dispatch an instruction to a warp.
- It takes 200 cycles to fetch from global memory.
- How many warps does it take to hide memory latency?

Block Size

- 8 by 8 block size
  - 768 / 64 = 12 > 8 blocks
  - Only 8 blocks = 512 threads are used
- 16 by 16 blocks
  - 768 / 256 = 3 blocks
  - All 3 blocks = 768 threads are used
- 32 by 32
  - 1024 threads cannot go into a SM

Tiling Size Effects



Control Flow

- SPMD program
  - Threads do things according to thread id.
- A warp should follow the same control path as offten as possible.

Granularity of Control Flow

- If threads with a warp take different control paths, they will be serialized.
  - Remember that the same instruction will be issued to all threads in the same block.
  - That means we will not have good speedup.
  - Therefore we should get all threads in a warp to go along the same control flow as much as possible.

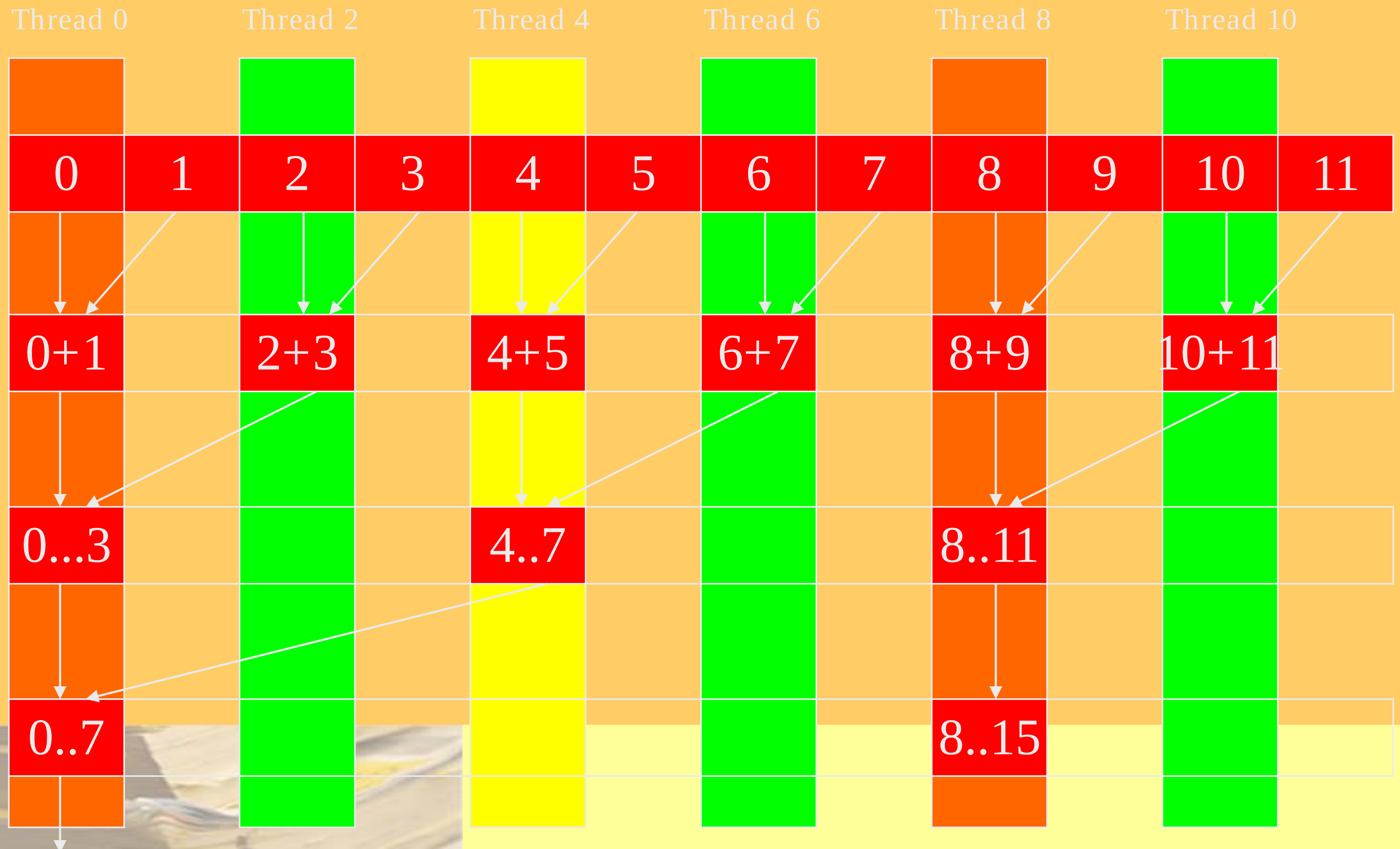
Examples

- if (threadIdx.x > 2)
  - Thread 0, 1, and 2 will take else.
  - 30 other threads take then.
  - Two groups will be serialized.
- If (threadIdx.x / groupsize > 2)
  - If groupsize is a multiple of 32, all threads in a warp go along the same control path.

Sum of N Numbers

- To add the numbers in a global array of N elements.
- Use a block of N threads to compute the sum.
- The computation is in log N phases.
  - In the first phase all even number threads with index i add the element in the odd number threads that have indices i + 1 to them.
  - The next phase all threads of indices multiple of 4 add the elements in the threads that have indices i + 2 to them.
  - We repeat this process until the first element has the sum of all elements.

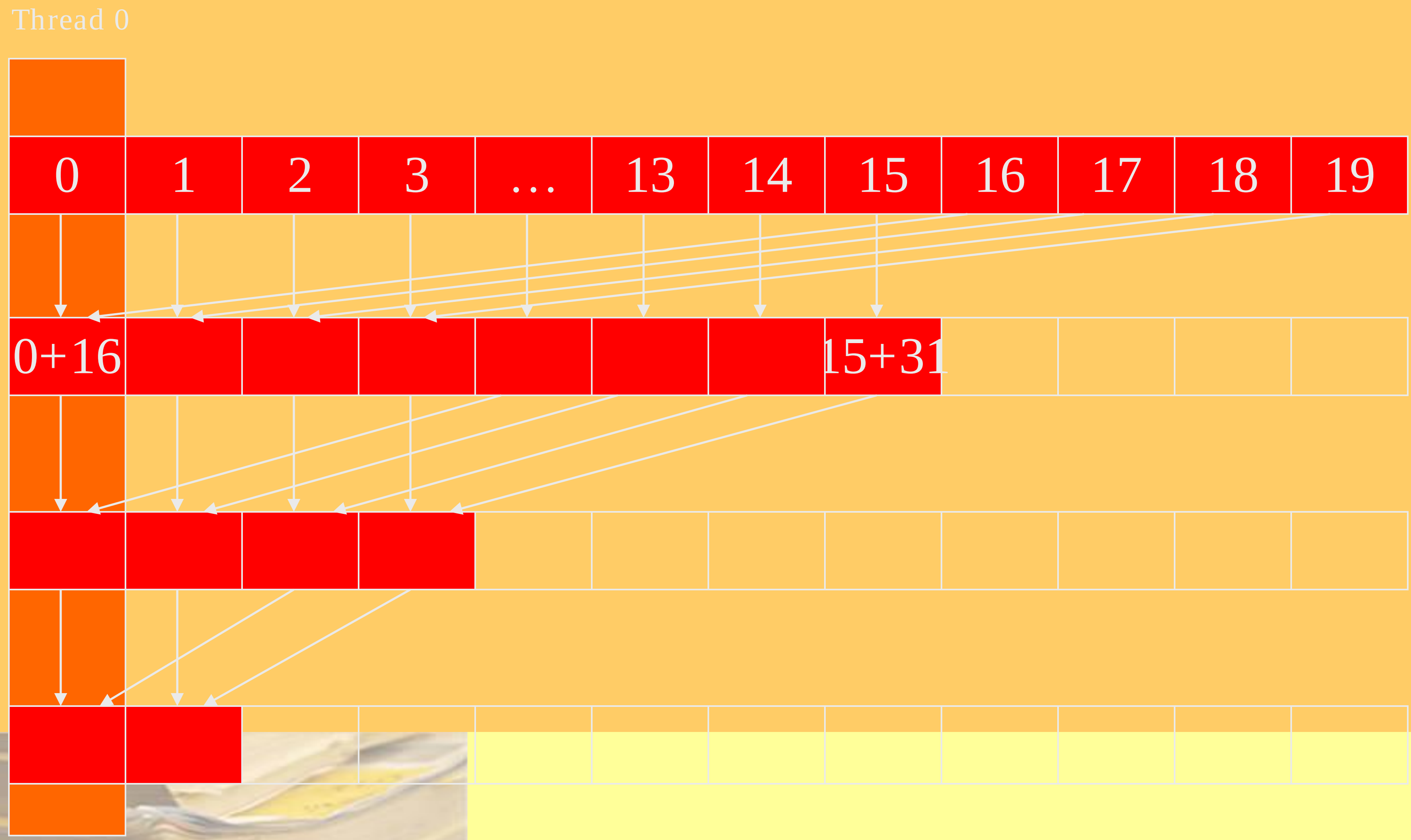
An Illustration



Reverse Order

- Add the element in decreasing strides.
- In the first iteration, those threads with indices smaller than N/2 will add the elements that have indices N/2 greater than them to the element they have.
- Then the stride reduces by half and we do it again.
- We repeat the process until the first element has the sum of all numbers.

An Illustration



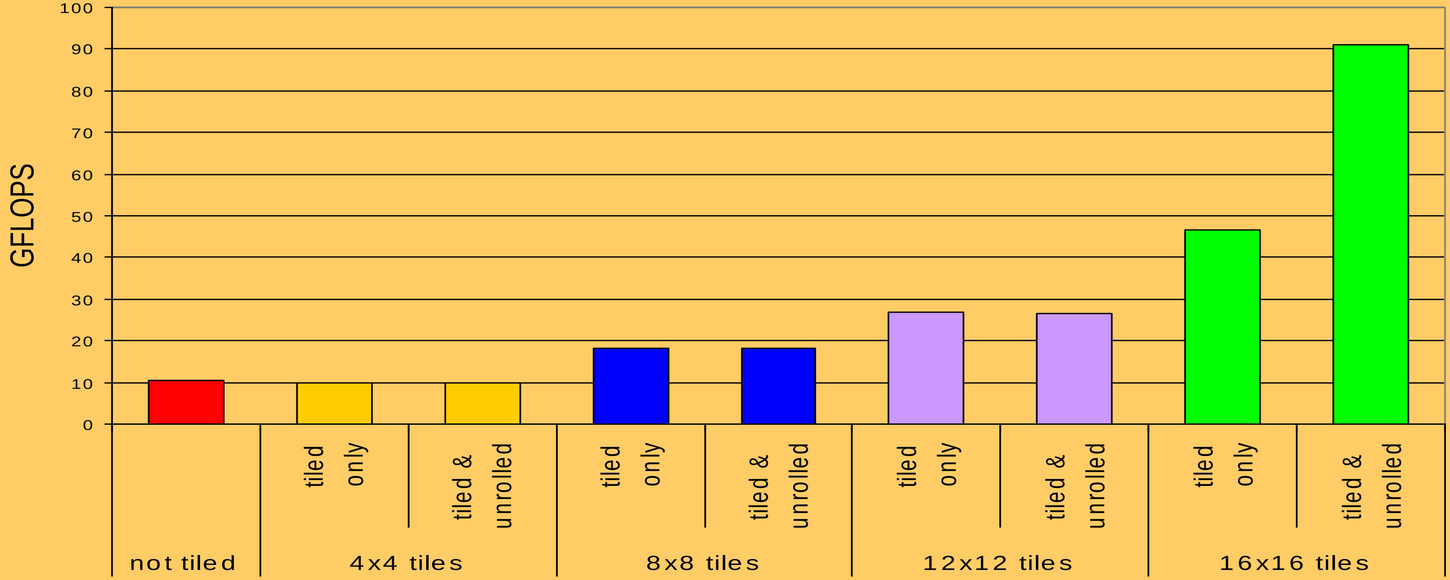
Memory Access

- A warp should access memory in horizontal (contiguous) order .
- If horizontal access is not possible, one should reduce the stride in rows.

Tiled Matrix Multiplication

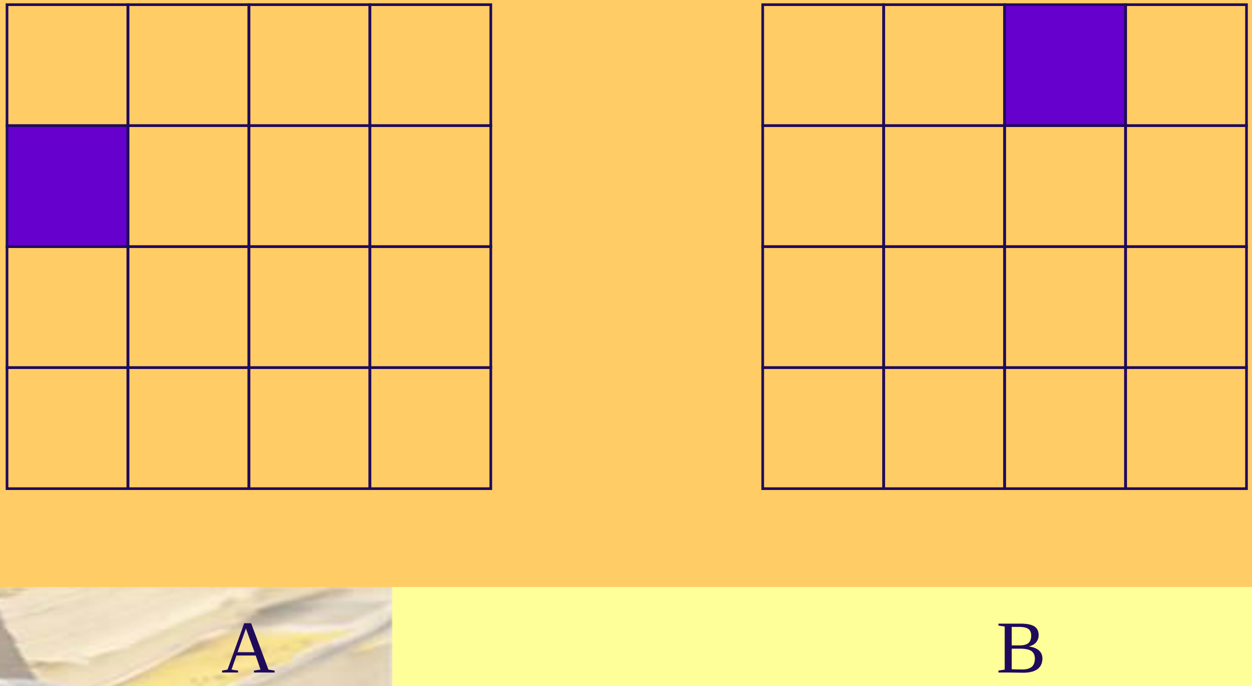
- In version 1 (using global memory only), the warps reference data horizontally (N) and vertically (N).
- In version 2 (using global and shared memory), the warps reference data horizontally (b) and vertically (b).

Tiling Size Effects



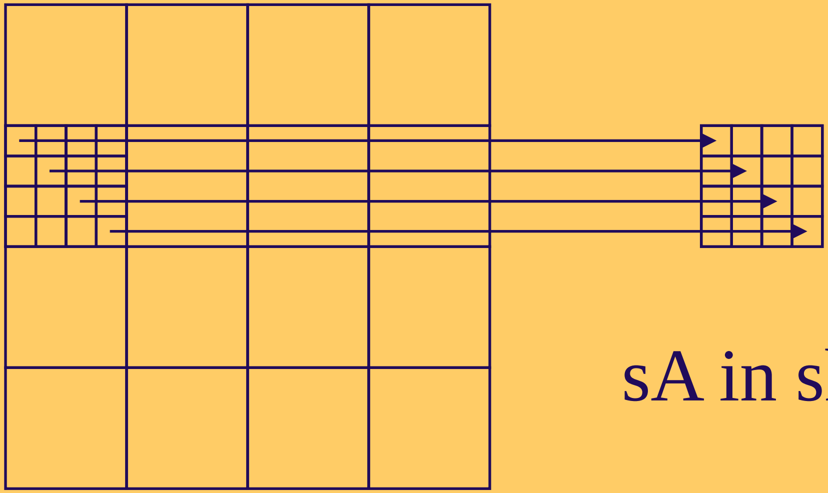
An Illustration

- First step



An Illustration

- Each thread moves an element in A.

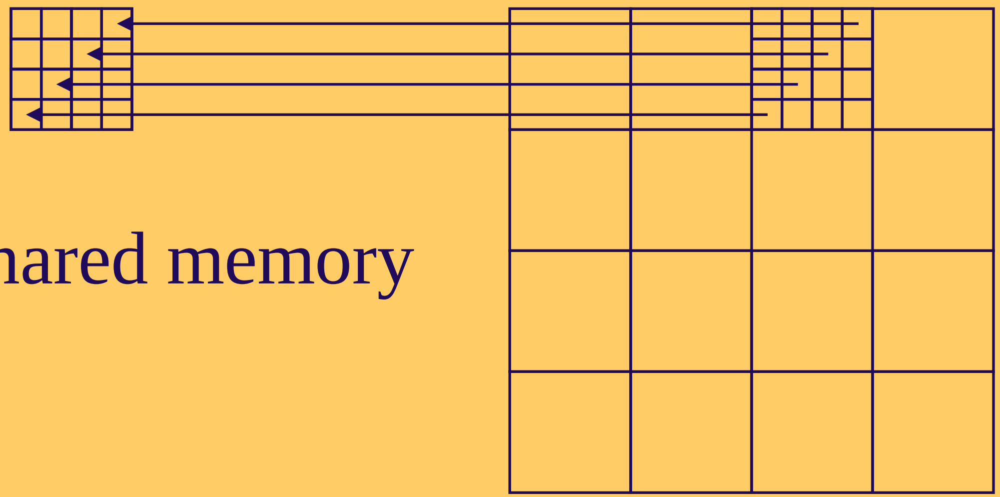


sA in shared memory

A in global memory

An Illustration

- Each thread moves an element in B

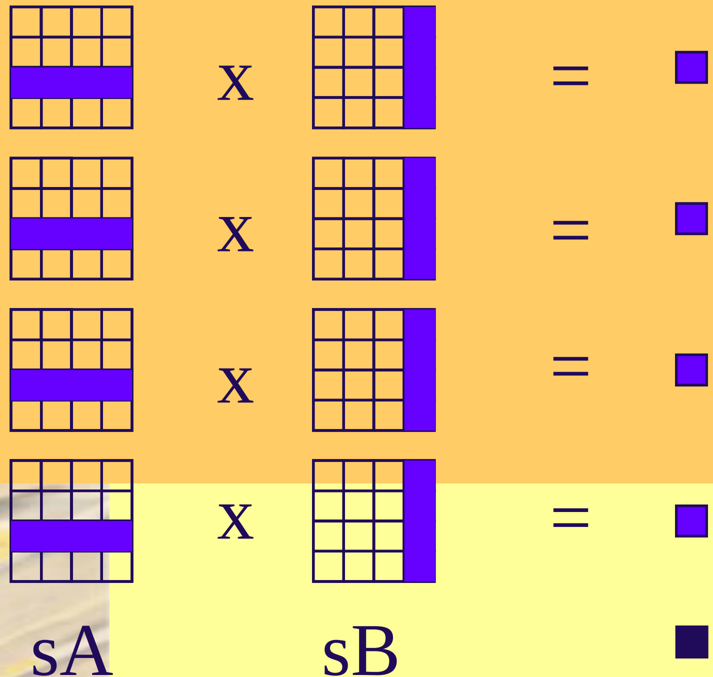


sB in shared memory

B in global memory

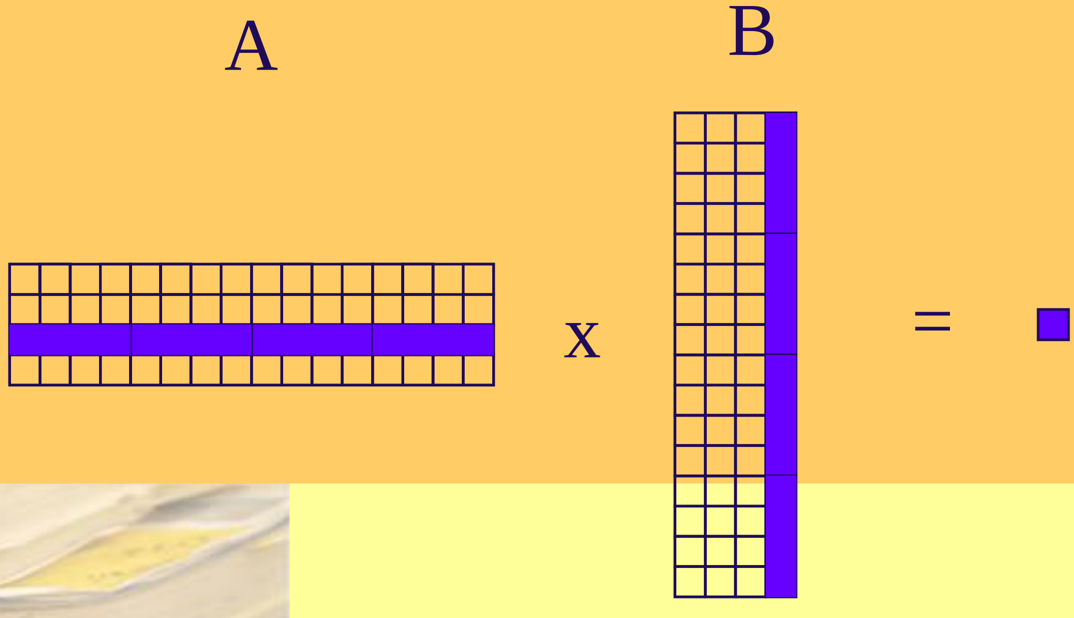
An Illustration

- Each thread computes an element in C using shared memory.



An Illustration

- Each thread computes an element in C using global memory.



Unroll

- Inline the most time consuming loop.
- for (k = 0; k < b; k++)
  - sum += sA[threadIdx.x][k] * sB[k][threadIdx.y];
- Change to the following.
  - sum += sA[threadIdx.x][0] * sB[0][threadIdx.y] + sA[threadIdx.x][1] * sB[1][threadIdx.y] + … + sA[threadIdx.x][7] * sB[7][threadIdx.y];

Advantage

- Without loop overhead
  - All indices are determined at compile time.
- Hopefully better function unit utilization.
- Larger amount of computation between jumps.

Double Buffering

- Use two buffers
  - One for ongoing memory fetch.
  - One for current computation.

An Illustration

- Load A and B into “current” shared memory
- Repeat
  - Load A and B into “next” shared memory.
  - Compute on “current” shared memory.
  - Switch the current and next memory.