# CSc 8530
# Parallel Algorithms

Spring 2019

March 26th, 2019

# Paper presentation guidelines

- The presentation will be graded based on the following criteria:
  1. Content mastery (40%): how well your group understands the paper's content. Based on both your presentation and how you answer questions from the audience.
  2. Preparedness (30%): slide quality; time management; smooth transitions between speakers, etc.
  3. Delivery (30%): quality of the oral presentation; effective use of visual aids, etc.
- **Note:** each person's grade will be **individual.**
- **Tips:**
  1. As a rule of thumb, allocate around 60-90 seconds per slide
  2. Aim for fewer words and more graphics in your slides
  3. Don't try to go over every detail in the paper (there is not enough time); focus on the big picture

# Matrix multiplication efficiency

```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
  int Width) {
  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;
  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;
  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k) {
      Pvalue += M[Row*Width+k]*N[k*Width+Col];
    }
    P[Row*Width+Col] = Pvalue;
  }

}
```

- The memory accesses of this program are dominated by the inner for-loop
    - Two global memory accesses
    - One addition and one multiplication
- Compute ratio of 1.0

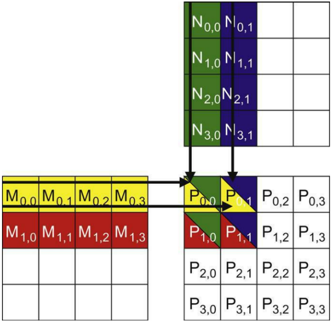
# CUDA memory types

| Variable declaration | Memory | Scope | Lifetime |
|---|---|---|---|
| Automatic variables other than arrays | Register | Thread | Kernel |
| Automatic array variables | Local | Thread | Kernel |
| `__device__ __shared__ int SharedVar;` | Shared | Block | Kernel |
| `__device__ int GlobalVar;` | Global | Grid | Application |
| `__device__ __constant__ int ConstVar;` | Constant | Grid | Application |

- In CUDA, we can declare variables so that reside in a specific type of memory
- **Scope** is the set of threads that have access to it
- **Lifetime** is how long the variable is maintained
  - e.g., the top three are initialized every time we call a kernel function

# Matrix multiplication revisited

- The colors indicate which threads need which parts of the input matrices
- The black arrows show the memory accesses of the first two threads, $P_{0,0}$ and $P_{0,1}$, from the first block
- They both need the same row, but different columns
- They access each needed element sequentially



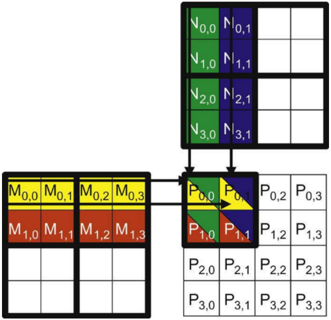
# Matrix multiplication revisited

Access order →

| | | | | |
|---|---|---|---|---|
| $\text{thread}_{0,0}$ | $\boxed{M_{0,0}} * N_{0,0}$ | $M_{0,1} * \boxed{N_{1,0}}$ | $M_{0,2} * N_{2,0}$ | $M_{0,3} * N_{3,0}$ |
| $\text{thread}_{0,1}$ | $\boxed{M_{0,0}} * N_{0,1}$ | $M_{0,1} * N_{1,1}$ | $M_{0,2} * N_{2,1}$ | $M_{0,3} * N_{3,1}$ |
| $\text{thread}_{1,0}$ | $M_{1,0} * N_{0,0}$ | $M_{1,1} * \boxed{N_{1,0}}$ | $M_{1,2} * N_{2,0}$ | $M_{1,3} * N_{3,0}$ |
| $\text{thread}_{1,1}$ | $M_{1,0} * N_{0,1}$ | $M_{1,1} * N_{1,1}$ | $M_{1,2} * N_{2,1}$ | $M_{1,3} * N_{3,1}$ |

- Above, we have the sequence of memory accesses for all threads, $T$, in block(0,0)
- e.g. both $T_{0,0}$ and $T_{0,1}$ access $M_{0,0}$
- Note how, in this example each element of $M$ and $N$ is accessed twice
    - If we only retrieve each element once, we reduce global memory accesses by half

# Matrix multiplication revisited




- Above, we divide $M$ and $N$ into 2×2 tiles
- We load one tile of $M$ and one of $N$ into the shared memory at a time
- In the simplest case, tile size equals block size
  - But it doesn't have to

# Matrix multiplication revisited

| | Phase 1 | | | Phase 2 | | |
|---|---|---|---|---|---|---|
| $\text{thread}_{0,0}$ | $\mathbf{M_{0,0}} \downarrow \text{Mds}_{0,0}$ | $\mathbf{N_{0,0}} \downarrow \text{Nds}_{0,0}$ | $\text{PValue}_{0,0} \mathrel{+}= \text{Mds}_{0,0}{*}\text{Nds}_{0,0} + \text{Mds}_{0,1}{*}\text{Nds}_{1,0}$ | $\mathbf{M_{0,2}} \downarrow \text{Mds}_{0,0}$ | $\mathbf{N_{2,0}} \downarrow \text{Nds}_{0,0}$ | $\text{PValue}_{0,0} \mathrel{+}= \text{Mds}_{0,0}{*}\text{Nds}_{0,0} + \text{Mds}_{0,1}{*}\text{Nds}_{1,0}$ |
| $\text{thread}_{0,1}$ | $\mathbf{M_{0,1}} \downarrow \text{Mds}_{0,1}$ | $\mathbf{N_{0,1}} \downarrow \text{Nds}_{1,0}$ | $\text{PValue}_{0,1} \mathrel{+}= \text{Mds}_{0,0}{*}\text{Nds}_{0,1} + \text{Mds}_{0,1}{*}\text{Nds}_{1,1}$ | $\mathbf{M_{0,3}} \downarrow \text{Mds}_{0,1}$ | $\mathbf{N_{2,1}} \downarrow \text{Nds}_{0,1}$ | $\text{PValue}_{0,1} \mathrel{+}= \text{Mds}_{0,0}{*}\text{Nds}_{0,1} + \text{Mds}_{0,1}{*}\text{Nds}_{1,1}$ |
| $\text{thread}_{1,0}$ | $\mathbf{M_{1,0}} \downarrow \text{Mds}_{1,0}$ | $\mathbf{N_{1,0}} \downarrow \text{Nds}_{1,0}$ | $\text{PValue}_{1,0} \mathrel{+}= \text{Mds}_{1,0}{*}\text{Nds}_{0,0} + \text{Mds}_{1,1}{*}\text{Nds}_{1,0}$ | $\mathbf{M_{1,2}} \downarrow \text{Mds}_{1,0}$ | $\mathbf{N_{3,0}} \downarrow \text{Nds}_{1,0}$ | $\text{PValue}_{1,0} \mathrel{+}= \text{Mds}_{1,0}{*}\text{Nds}_{0,0} + \text{Mds}_{1,1}{*}\text{Nds}_{1,0}$ |
| $\text{thread}_{1,1}$ | $\mathbf{M_{1,1}} \downarrow \text{Mds}_{1,1}$ | $\mathbf{N_{1,1}} \downarrow \text{Nds}_{1,1}$ | $\text{PValue}_{1,1} \mathrel{+}= \text{Mds}_{1,0}{*}\text{Nds}_{0,1} + \text{Mds}_{1,1}{*}\text{Nds}_{1,1}$ | $\mathbf{M_{1,3}} \downarrow \text{Mds}_{1,1}$ | $\mathbf{N_{3,1}} \downarrow \text{Nds}_{1,1}$ | $\text{PValue}_{1,1} \mathrel{+}= \text{Mds}_{1,0}{*}\text{Nds}_{0,1} + \text{Mds}_{1,1}{*}\text{Nds}_{1,1}$ |

time $\longrightarrow$

- All the threads collaborate to load individual elements from the current tile
- Once elements are loaded onto shared memory, they can be accessed by multiple threads

# Matrix multiplication revisited

| | Phase 1 | | | Phase 2 | | |
|---|---|---|---|---|---|---|
| $thread_{0,0}$ | $\mathbf{M}_{0,0}$ ↓ $Mds_{0,0}$ | $\mathbf{N}_{0,0}$ ↓ $Nds_{0,0}$ | $PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$ | $\mathbf{M}_{0,2}$ ↓ $Mds_{0,0}$ | $\mathbf{N}_{2,0}$ ↓ $Nds_{0,0}$ | $PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$ |
| $thread_{0,1}$ | $\mathbf{M}_{0,1}$ ↓ $Mds_{0,1}$ | $\mathbf{N}_{0,1}$ ↓ $Nds_{1,0}$ | $PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$ | $\mathbf{M}_{0,3}$ ↓ $Mds_{0,1}$ | $\mathbf{N}_{2,1}$ ↓ $Nds_{0,1}$ | $PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$ |
| $thread_{1,0}$ | $\mathbf{M}_{1,0}$ ↓ $Mds_{1,0}$ | $\mathbf{N}_{1,0}$ ↓ $Nds_{1,0}$ | $PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$ | $\mathbf{M}_{1,2}$ ↓ $Mds_{1,0}$ | $\mathbf{N}_{3,0}$ ↓ $Nds_{1,0}$ | $PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$ |
| $thread_{1,1}$ | $\mathbf{M}_{1,1}$ ↓ $Mds_{1,1}$ | $\mathbf{N}_{1,1}$ ↓ $Nds_{1,1}$ | $PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$ | $\mathbf{M}_{1,3}$ ↓ $Mds_{1,1}$ | $\mathbf{N}_{3,1}$ ↓ $Nds_{1,1}$ | $PValue_{1,1}$+= $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$ |

time →

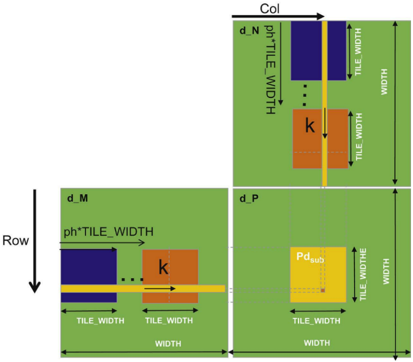
- We now have to compute the dot product in **phases**
  - We accumulate the partial dot product after every phase
- We progressively load parts of the rows and columns into smaller arrays Mds and Nds
  - These array live in shared memory
  - We can reuse them every time a new tile is loaded
    - Memory accesses exhibit **locality**

# Matrix multiplication revisited

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
    int Width) {

    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    // Identify the row and column of the d_P element to work on
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0;
    // Loop over the d_M and d_N tiles required to compute d_P element
    for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {

      // Collaborative loading of d_M and d_N tiles into shared memory
      Mds[ty][tx] = d_M[Row*Width + ph*TILE_WIDTH + tx];
      Nds[ty][tx] = d_N[(ph*TILE_WIDTH + ty)*Width + Col];
      __syncthreads();

      for (int k = 0; k < TILE_WIDTH; ++k) {
        Pvalue += Mds[ty][k] * Nds[k][tx];
      }
      __syncthreads();
    }
    d_P[Row*Width + Col] = Pvalue;
  }
```

- `__shared__` arrays `Mds` and `Nds` are common to the *block*
- `__syncthreads();` ensures that all the elements in a tile have been loaded before the next phase of the dot product

# Matrix multiplication revisited




- Our memory accesses are reduced by a factor of TILE_WIDTH
- For 16×16 tiles, this yields a compute ratio of 16
  - e.g., with 150 GB/s bandwidth, we can achieve $(150/4)16 =$ 600 GFLOPS

# Prefix sums revisited

- We will now revisit the prefix sums problem
  - In the context of GPUs
- As before, this is a *model* problem
  - Also called a *pattern* in software development
  - The techniques used to solve it are applicable to a wide variety of other problems
- We will investigate three different types of kernel functions:
  - Kogge-Stone
  - Brent-Kung
  - two-phase hybrid
- Each involves different computational tradeoffs

# Prefix sums

- Let $S = \{x_1, x_2, \ldots, x_n\}$ be an $n$-element set
- Let $*$ be a binary associate operation (e.g., sum or product)
- A **prefix sum** is the partial sum defined by:

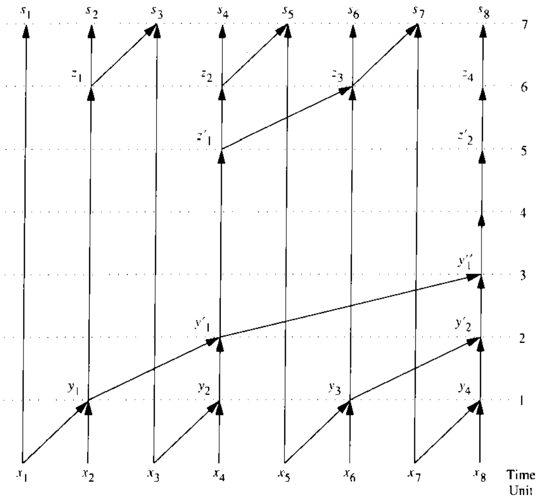
$$s_i = x_1 * x_2 * \ldots * x_i, 1 \leq i \leq n$$

- The **prefix sums** are the $n$ partial products $s_1$ to $s_n$
- A trivial sequential algorithm can compute $s_i$ from $s_{i-1}$ as $s_i = s_{i-1} * x_i$
  - Clearly, this algorithm is $O(n)$

# Prefix sums

- We can use a balanced binary tree to compute the prefix sums in $O(\log{(n)})$
- We compute pairwise $*$ operations during the forward pass
- Each internal node will hold the sum of the elements stored in the leaves of its subtree
- During the backward pass, we compute the prefix sums at each level of the tree

# Balanced binary trees



# Recursive vs. non-recursive versions

**ALGORITHM 2.1**
**(Prefix Sums)**
**Input:** *An array of* $n = 2^k$ *elements* $(x_1, x_2, \ldots, x_n)$, *where* $k$ *is a nonnegative integer.*
**Output:** *The prefix sums* $s_i$, *for* $1 \le i \le n$.
**begin**
  *1.* **if** $n = 1$ **then** {*set* $s_1 := x_1$; **exit**}
  *2.* **for** $1 \le i \le n/2$ **pardo**
    *Set* $y_i := x_{2i-1} * x_{2i}$
  *3. Recursively, compute the prefix sums of* $\{y_1, y_2, \ldots, y_{n/2}\}$, *and store them in* $z_1, z_2, \ldots, z_{n/2}$.
  *4.* **for** $1 \le i \le n$ **pardo**
    $\{i$ *even* : *set* $s_i := z_{i/2}$
    $i = 1$ : *set* $s_1 := x_1$
    $i$ *odd* $> 1$ : *set* $s_i := z_{(i-1)/2} * x_i\}$
**end**

## Recursive

**ALGORITHM 2.2**
**(Nonrecursive Prefix Sums)**
**Input:** *An array* $A$ *of size* $n = 2^k$, *where* $k$ *is a nonnegative integer.*
**Output:** *An array* $C$ *such that* $C(0, j)$ *is the* $j$*th prefix sum, for* $1 \le j \le n$.
**begin**
  *1.* **for** $1 \le j \le n$ **pardo**
    *Set* $B(0, j) := A(j)$
  *2.* **for** $h = 1$ **to** $\log n$ **do**
    **for** $1 \le j \le n/2^h$ **pardo**
      *Set* $B(h, j) := B(h-1, 2j-1) * B(h-1, 2j)$
  *3.* **for** $h = \log n$ **to** $0$ **do**
    **for** $1 \le j \le n/2^h$ **pardo**
      $\{j$ *even* : *Set* $C(h, j) := C\left(h+1, \frac{j}{2}\right)$
      $j = 1$ : *Set* $C(h, 1) := B(h, 1)$
      $j$ *odd* $> 1$ : *Set* $C(h, j) := C\left(h+1, \frac{j-1}{2}\right) * B(h, j)\}$
**end**

## Non-recursive

# Parallel scans

- Parallel scans (i.e, a more general version of prefix sums) are often used to convert seemingly sequential operations into parallel ones
- Most recursive functions can be formulated as parallel scans
- In general, given an input set $S = \{x_1, x_2, \ldots, x_n\}$ and a binary operation $\oplus$ our output set is:

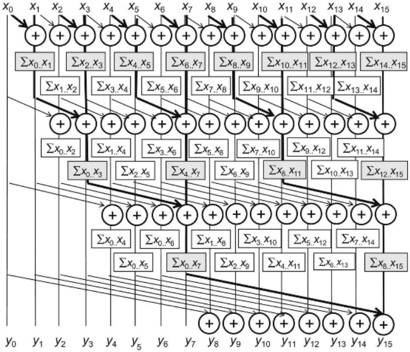
$$\{\,\{x_1\}, \{x_1 \oplus x_2\}, \ldots, \{x_1 \oplus x_2 \oplus \ldots x_{n_1} \oplus x_n\}\,\}$$

# Kogge-Stone – a simple parallel scan

- As noted before, we can compute a parallel scan by *reducing* the input elements
  - That is, merging partial results using a binary tree
- The Kogge-Stone algorithm is one of the simplest ways to build this tree
- At iteration 0, we assume position $X[i]$ in our input array contains element $x_i$
- At iteration $n$, $X[i]$ will contain the sum of $2^n$ elements leading to $i$ (including $x_i$):
  - e.g., for $n = 2$, we have:

$$X[i] = x_{i-3} + x_{i-2} + x_{i-1} + x_i$$

# Kogge-Stone – illustration




- The above illustrates the algorithm for a 16-element array
- At iteration $j$, each element adds its current value with the value of the element that is $2^j$ steps before it
  - If this element is out of bounds, then we stop computing values for that position

# Kogge-Stone – code

```
__global__ void Kogge-Stone_scan_kernel(float *X, float *Y,
  int InputSize) {

  __shared__ float XY[SECTION_SIZE];

  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < InputSize) {
    XY[threadIdx.x] = X[i];
  }

  // the code below performs iterative scan on XY
  for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
    __syncthreads();
    if (threadIdx.x >= stride)XY[threadIdx.x] += XY[threadIdx.x-stride];
  }

  Y[i] = XY[threadIdx.x];

}
```

- The above code computes Kogge-Stone for a **section** of the array that is small enough to fit in a block
  - Each thread is responsible for one element of the output array
- We will see how to combine multiple blocks later

# Kogge-Stone – speed and work efficiency

- We will now analyze the performance of the previous kernel
- All threads execute for $\log(n)$ steps, where $n$ is SECTION_SIZE
  - Why not array size?
- In each iteration $j$, the number of *inactive* threads is equal to the stride size, $2^j$
- Thus, the total work is:

$$
\begin{aligned}
W &= \sum_{j=0}^{\log(n)} n - 2^j \\
&= n\log(n) - (n-1) \\
&= O(n\log(n))
\end{aligned}
$$

- Compare to the sequential algorithm: $O(n)$