# CSc 8530
# Parallel Algorithms

Spring 2019

April 16th, 2019

# End-of-semester programmatic details

- There will be one more homework
  - Out Thursday, April 18th
  - Due Friday, April 26th
  - Remember that the lowest homework grade will be dropped
- Project presentations will be **5 minutes**
  - On the last day of class
  - **One summary slide only**
    - Think of it as a mini-poster
  - Email me your summary slide **before 10:00 am on Thursday, April 25th**
- Final project report due on **Friday, May 3rd at 11:59pm.**

# 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:

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

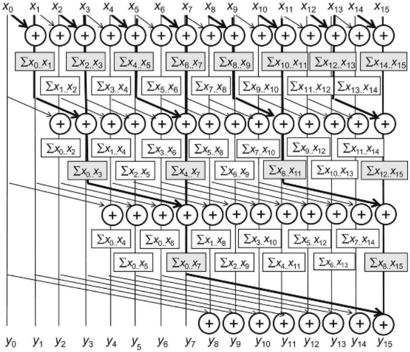
- The **prefix sums** are the $n$ partial products $s_1$ to $s_n$
- A trivial sequential algorithm can compute $s_i$ from $s_{i-1}$ as $s_i = s_{i-1} * x_i$
  - Clearly, this algorithm is $O(n)$

# 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:

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

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

# Kogge-Stone – speed and work efficiency

- The running time is $T(n) = O(\log(n))$
- So the speedup with $n$ processors is:

$$S_n(n) = \frac{O(n)}{O(\log(n))}$$

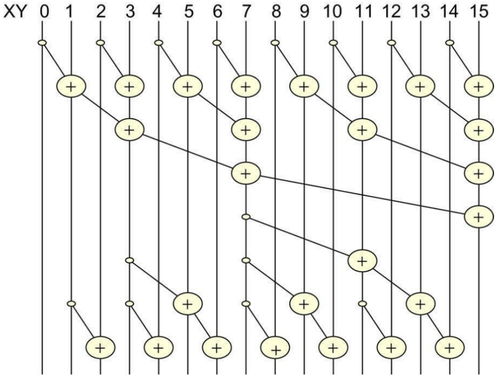
- e.g. for $n = 512$, the speedup is:

$$\begin{aligned} S_{512}(512) &= \frac{512}{9} \\ &\approx 56.9 \end{aligned}$$

# Brent-Kung: a more efficient kernel

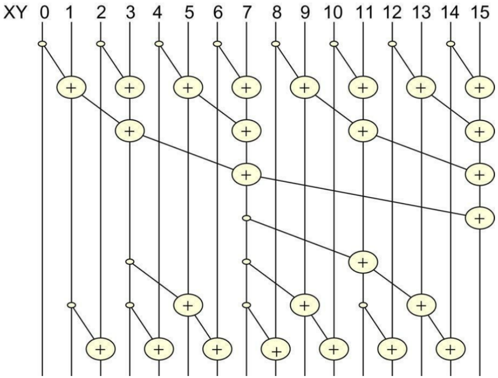
- The Kogge-Stone kernel is simple, but work-inefficient
- The $\log(n)$ factor can be significant in real-world applications
  - e.g., for $n = 512$, we will need about 8 times the resources compared to $O(n)$ work
- We will now study the more efficient, Brent-Kung kernel
  - Intuitively, we reuse intermediate calculations more effectively
  - A form of dynamic programming

# Brent-Kung: illustration




- We use the minimal number of operations to produce the sum
- In the first step, we only update odd elements
- At step $i$, we only update elements at positions $2^i(n-1)$

# Brent-Kung: illustration




- Above, the total number of operations in the first half is $8 + 4 + 2 + 1 = 15 = O(n)$
- In general, we do $(n/2) + (n/4) + (n/8)...2 + 1 = n - 1$ operations

# Brent-Kung: second half

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x_0$ | $x_0.x_1$ | $x_2$ | $x_0.x_3$ | $x_4$ | $x_4.x_5$ | $x_6$ | $x_0.x_7$ | $x_8$ | $x_8..x_9$ | $x_{10}$ | $x_8.x_{11}$ | $x_{12}$ | $x_{12}..x_{13}$ | $x_{14}$ | $x_0.x_{15}$ |
| | | | | | | | | | | | $x_0..x_{11}$ | | | | |
| | | | | | $x_0..x_5$ | | | | $x_0..x_9$ | | | | $x_0..x_{13}$ | | |

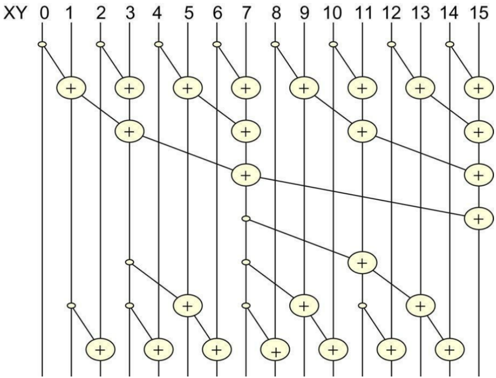
- In the second half of the algorithm, we distribute the partial sums as quickly as possible
- Above, the first row shows the partial sums available at each element after the first half
- In this example, XY[0], XY[7], and XY[15] already contain their final answers
- Thus, no other element needs a partial sum that is more than *four* elements away

# Brent-Kung: second half

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x_0$ | $x_0.x_1$ | $x_2$ | $x_0.x_3$ | $x_4$ | $x_4.x_5$ | $x_6$ | $x_0.x_7$ | $x_8$ | $x_8..x_9$ | $x_{10}$ | $x_8.x_{11}$ | $x_{12}$ | $x_{12}..x_{13}$ | $x_{14}$ | $x_0.x_{15}$ |
| | | | | | | | | | | | $x_0..x_{11}$ | | | | |
| | | | | | $x_0..x_5$ | | | | $x_0..x_9$ | | | | $x_0..x_{13}$ | | |

- Here, only XY[11] need a value four positions behind
  - At XY[7]
  - After updating this value, it can be used to update elements XY[12] to XY[14]
- Elements XY[5], XY[9], and XY[13] need a value from *two* positions behind
- The remaining elements need a value that is one element behind

# Brent-Kung: second half




- Note how we avoid computing intermediate values until we've accumulated the information in earlier positions
- We minimize duplicate work

# Brent-Kung: pseudocode

```
__global__ void Brent_Kung_scan_kernel(float *X, float *Y,
  int InputSize) {

  __shared__ float XY[SECTION_SIZE];
  int i = 2*blockIdx.x*blockDim.x + threadIdx.x;
  if (i < InputSize) XY[threadIdx.x] = X[i];
  if (i+blockDim.x < InputSize) XY[threadIdx.x+blockDim.x] = X[i+blockDim.x];

  for (unsigned int stride = 1; stride <= blockDim.x; stride *= 2) {
    __syncthreads();
    int index = (threadIdx.x+1) * 2* stride -1;
    if (index < SECTION_SIZE) {
      XY[index] += XY[index - stride];
    }
  }

  for (int stride = SECTION_SIZE/4; stride > 0; stride /= 2) {
    __syncthreads();
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index + stride < SECTION_SIZE) {
      XY[index + stride] += XY[index];
    }
  }

  __syncthreads();
  if (i < InputSize) Y[i] = XY[threadIdx.x];
  if (i+blockDim.x < InputSize) Y[i+blockDim.x] = XY[threadIdx.x+blockDim.x];
}
```

# Brent-Kung: analysis

- In our running example, the total number of operations in the second half is $(2-1)+(4-1)+(8-1)$
  - For the first half it is $8+4+2+1$
- In general, we have

$$\begin{aligned} W(n) &= [(2-1)+(4-1)+\ldots+(n/4-1)+(n/2-1)] \\ &\quad + [n/2+n/4+\ldots 2+1] \\ &= n-1-\log(n)+(n-1) \\ &= O(n) \end{aligned}$$

- Thus, Brent-Kung is (theoretically) weakly optimal

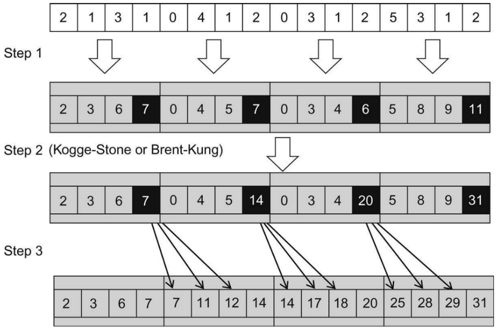
# Brent-Kung: analysis

- Theoretically, Brent-Kung is weakly optimal
- In CUDA, the difference between Kogge-Stone and Brent-Kung is much smaller
- Brent-Kung uses $n/2$ threads
  - The maximum needed at any given step
- The number of *active* threads drops much quicker in Brent-Kung than Kogge-Stone
- However, the inactive threads still consume GPU resources (e.g., SMs, memory)
- The real-world work efficiency is closer to $(n/2)(2\log(n) - 1) = O(n\log(n))$
  - Asymptotically identical to Kogge-Stone
  - Hence, their *cost* is the same

# An even more work efficient kernel

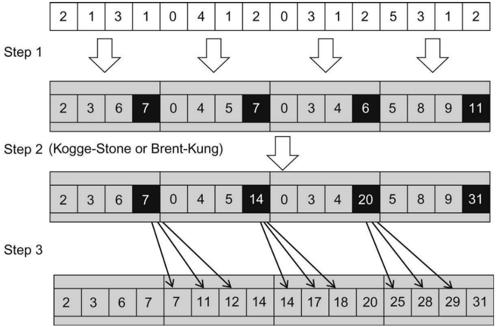
- We can achieve a higher work-efficiency than Brent-Kung
  - By adding a phase of fully independent scans on subsections of the input
- The number of subsections will be the same as the number of threads in a block
  - i.e., one subsection per thread
- During the first phase, each thread will sequentially scan its subsection
- The threads first cooperatively load the values into memory
  - Using carpooling
- At the end of the first phase, the last element in each subsection will contain the partial sum of that subsection

# An even more efficient kernel



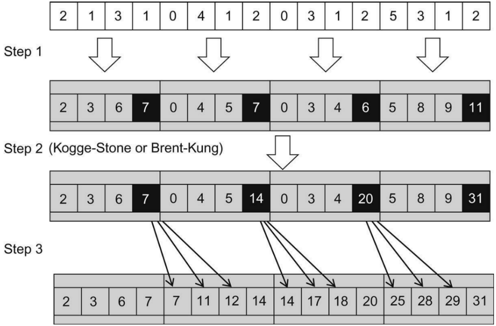

- In Step 1, we sequentially scan each subsection
- In Step 2, we use Kogge-Stone or Brent-Kung, but only for the elements in **black** (i.e., last element of each subsection)

# An even more efficient kernel




- In Step 3, we add the updated last element of each subsection to the elements of the subsequent subsection
- This approach uses fewer total operations in practice

# An even more efficient kernel




- With $n$ elements and $t$ threads, we do:
  - Phase 1: $n - 1$ operations
  - Phase 2: $t \log(t)$ operations (Kogge-Stone)
  - Phase 3: $n - t$ operations
- If $t = O(\log(n))$, then $W(n) = O(n)$, **in practice**

# Convolution

- Convolution is a fundamental data processing operation
  - It is ubiquitous in signal processing, image processing, and probability, data science, etc.
  - It can be defined for any number of dimensions
- Intuitively, it corresponds to sliding one function (the **kernel**) along another (the *input*) and adding the product of the two functions at each location
  - In other words, it is a weighted sum that depends on the relative offset of the two functions
- Typically, the kernel will be a spatially bounded function
  - i.e., it will only have non-zero values for a narrow range
- This sliding process allows us to identify meaningful regions in the input function
  - Essentially, regions that are similar to the kernel

# Convolution

- Mathematically, convolution is defined as (continuous):

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

  and discrete:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f(m)g(n-m)$$

- In both cases, you can think of the dummy variable (either $\tau$ or $m$) as the index of a for-loop
- In a computer the summation doesn't extend to infinity:

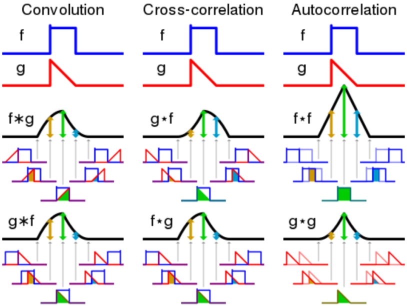
$$(f * g)[n] = \sum_{m=m_{\text{min}}}^{m_{\text{max}}} f(m)g(n-m)$$

# Convolution

- Technically, people will often refer to convolution when they really mean cross-correlation:

$$(f \star g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t+\tau)d\tau$$

  - Note how the dummy variable $\tau$ is added, not subtracted
  - Subtracting $\tau$ flips the kernel (see drawing)
- For symmetric kernels (a common case), the two operations are identical




From Wikipedia, CC BY-SA 3.0