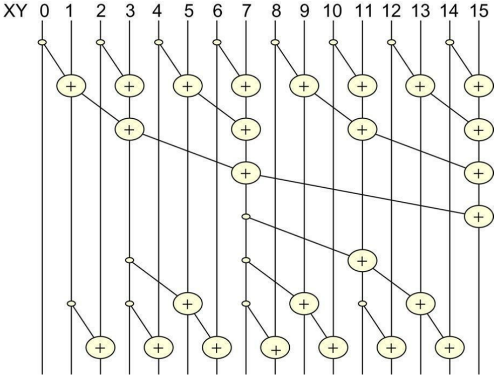
CSc 8530
Parallel Algorithms

Spring 2019

April 18th, 2019

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: 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

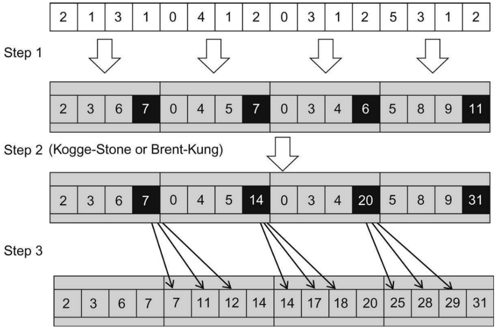
Brent-Kung: pseudocode

```
__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

- 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 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 τ or m) as the index of a for-loop
- In a computer the summation doesn't extend to infinity:

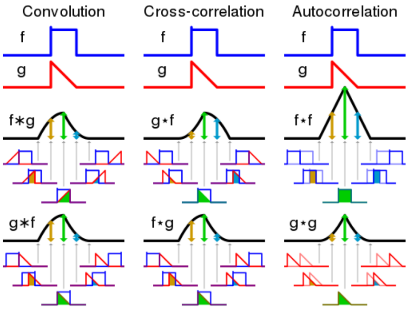
$$(f * g)[n] = \sum_{m=m_{\text{min}}}^{m_{\text{max}}} f(m)g(n-m)$$

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 τ is added, not subtracted
  - Subtracting τ flips the kernel (see drawing)
- For symmetric kernels (a common case), the two operations are identical

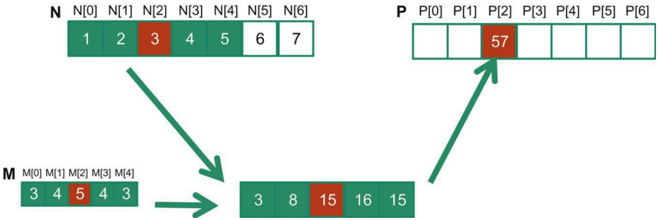


From Wikipedia, CC BY-SA 3.0

Convolution in GPUs

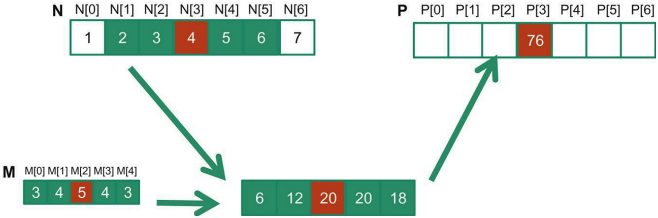
- Computing a convolution requires a large number of products and additions
- Both operations are linear, so they can be computed *independently*
  - And hence are good targets for parallel processing
- Here, we will refer to convolution kernels as *masks*
  - To avoid confusion with CUDA kernel functions
- We will first study 1D convolutions (e.g., for audio processing) and then look at 2D ones (for image processing)

1D convolution example



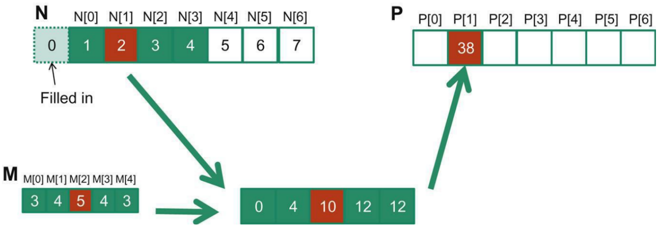
- Here, M and N are the mask and input, resp.
- The value for $N[2]$ is given by the weighted sum of it and its two neighbors on either side
- To calculate $N[3]$ we would slide M over by one to the right
- Typically, masks have odd number of elements so that sum is symmetric around the current element
    - Except for border elements, as we saw in the image blurring example

1D convolution example



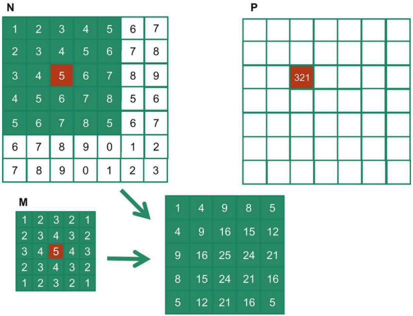
- To calculate $N[3]$ we slide M over by one to the right
- Typically, masks have odd number of elements so that sum is symmetric around the current element

1D convolution example



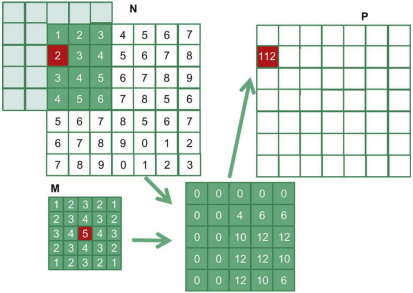
- We typically pad border elements with zeros
- In some applications (e.g., image processing) we can also replicate the border values or their mean
  - So that the hallucinated values have the same range as the real data
  - This approach can introduce artifacts, though
- My rule of thumb is that convolution is not meaningful for border elements

2D convolution example



- Two-dimensional convolution is conceptually the same as 1D
- We just shift the 2D kernel in both the x and y directions
- In sequential code, this corresponds to a nested for-loop
  - Or a linearized single for-loop

2D convolution example



- As in the 1D case, we have to hallucinate values for border elements

1D convolution: simple CUDA code

```
__global__ void convolution_1D_basic_kernel(float *N, float *M, float *P,
  int Mask_Width, int Width) {

  int i = blockIdx.x*blockDim.x + threadIdx.x;

  float Pvalue = 0;
  int N_start_point = i - (Mask_Width/2);
  for (int j = 0; j < Mask_Width; j++) {
    if (N_start_point + j >= 0 && N_start_point + j < Width) {
      Pvalue += N[N_start_point + j]*M[j];
    }
  }
  P[i] = Pvalue;

}
```

- The above code implicitly handles border elements
- It uses the local variable `Pvalue` (stored in a register) to reduce global memory accesses

1D convolution: simple CUDA code

```
__global__ void convolution_1D_basic_kernel(float *N, float *M, float *P,
  int Mask_Width, int Width) {

  int i = blockIdx.x*blockDim.x + threadIdx.x;

  float Pvalue = 0;
  int N_start_point = i - (Mask_Width/2);
  for (int j = 0; j < Mask_Width; j++) {
    if (N_start_point + j >= 0 && N_start_point + j < Width) {
      Pvalue += N[N_start_point + j]*M[j];
    }
  }
  P[i] = Pvalue;

}
```

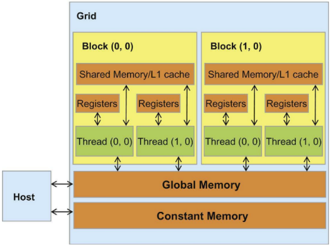
- Despite that, the kernel function is not memory efficient
- It has a compute ratio of 1.0
  - Since we access the N and M arrays directly from global memory every time

Improving memory bandwidth

- In typical applications, the mask has the following properties:
  1. It is small
     - Often on the order of tens of elements in 1D
  2. It is not modified during the convolution
  3. All threads needs to access all the mask elements
- Moreover, all threads access the mask *in the same order*
- These properties make it an excellent candidate for **constant memory** and **caching**

Improving memory bandwidth

- As we had discussed earlier in the course, CUDA allows you to declare a variable as `__constant__`
- The variable should be declared *outside* of any kernel functions
  - Since it is visible to *all* kernel functions
- Once a variable `M_h` has been declared in the host, we copy it to the device as follows:
  `cudaMemcpyToSymbol(M_d,M_h,Mask_Width*sizeof(float));`
  - Note that we use a different function (not `cudaMemcpy`)



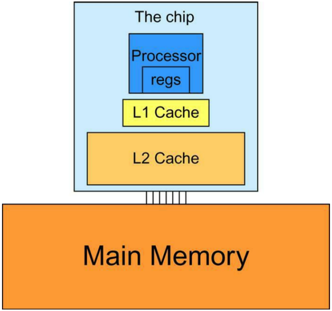
1D convolution: improved CUDA code

```
__global__ void convolution_1D_ba sic_kernel(float *N, float *P, int Mask_Width,
  int Width) {

  int i = blockIdx.x*blockDim.x + threadIdx.x;

  float Pvalue = 0;
  int N_start_point = i - (Mask_Width/2);
  for (int j = 0; j < Mask_Width; j++) {
    if (N start_point + j >= 0 && N_start_point + j < Width) {
      Pvalue += N[N_start_point + j]*M[j];
    }
  }
  P[i] = Pvalue;

}
```

- The kernel code with a constant mask is almost identical
  - Because constant variables are declared outside the scope of any function
- The only difference is that `M` is now a global (i.e., constant) variable, not a input parameter
  - From the point of the view of the kernel function, whether a variable is `global` or `constant` is immaterial

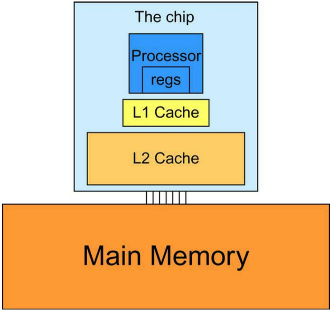
Memory caching

- Modern CPUs have multiple levels of *caches*
- Caches are transparent to a program
  - You don't know if an item was retrieved from a cache or not
  - Unlike shared variables in a GPU
- The closer to the processor itself, the faster and smaller the cache
  - The standard memory trade-off



Memory caching

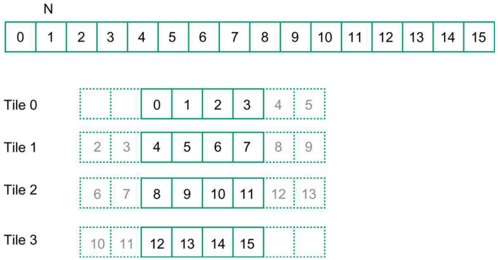
- Determining which variables to cache (and thus save time on future memory accesses) is more an art than a science
    - You need to predict what an algorithm will do in the future
    - Not possible in general for arbitrary code (see, e.g., the halting problem)
- If a cached variable is changed, it must be updated in main memory too
    - The *cache coherence problem*
- Constant variables are *not* changed, so they can be preferentially cached (in L1 if possible)
    - In principle, we can eliminate *all* global memory accesses



Tiled convolution

- In a tiled algorithm, multiple threads collaborate to load commonly used values into shared memory
  - Recall the matrix multiplication example
- We will now see how to use tiling to improve our convolution algorithm
- In the following discussion, we will assume that we have 1024 threads in a block
  - So we can process 1024 items simultaneously
  - We will refer to this set as an *output tile*

Tiled convolution – example



- We split our input array into four tiles of four threads each
  - In a real application, the tiles would be bigger ($\geq$ 32 threads)
- Convolution is a *local* operation, so most of the outputs can be computed using a single tile
  - The shaded elements are needed to compute the output values for that tile, but are not part of the tile itself
- Note that the mask M is in constant memory

Tiled convolution

- The simplest strategy for using tiling is to have all the threads in a block load their corresponding values into shared memory
  - Plus the bordering elements needed (i.e., the shaded locations in the drawing)
- For simplicity, assume the mask width is $2n + 1$
  - 5 in the example, with $n = 2$
- In general, to calculate $P[i]$, we need the values: $N[i-n], N[i-(n-1)], \ldots, N[i], \ldots, N[i+(n-1)], N[i+n]$
  - Note that we need fewer for border elements
- Thus, in our example tile $P[4:7]$ needs to load elements $N[2:9]$ onto shared memory
  - Note that some elements (e.g., $N[3]$) will be loaded multiple times onto different shared memories
    - Commonly referred to as *halo cells*

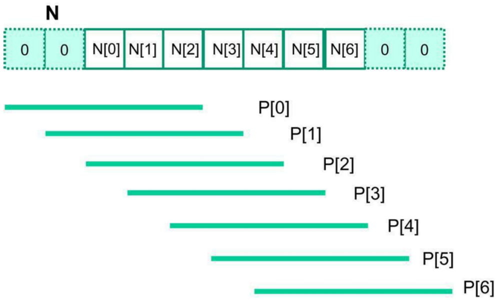
Tiled convolution – code

```
__global__ void convolution_1D_tiled_kernel(float *N, float *P, int Mask_Width,
  int Width) {

  int i = blockIdx.x*blockDim.x + threadIdx.x;
  __shared__ float  N_ds[TILE_SIZE + MAX_MASK_WIDTH - 1];

  int n = Mask_Width/2;

  int halo_index_left = (blockIdx.x - 1)*blockDim.x + threadIdx.x;
  if (threadIdx.x >= blockDim.x - n) {
    N_ds[threadIdx.x - (blockDim.x - n)] =
      (halo_index_left < 0) ? 0 : N[halo_index_left];
  }

  N_ds[n + threadIdx.x] = N[blockIdx.x*blockDim.x + threadIdx.x];

  int halo_index_right = (blockIdx.x + 1)*blockDim.x + threadIdx.x;
  if (threadIdx.x < n) {
    N_ds[n + blockDim.x + threadIdx.x] =
      (halo_index_right >= Width) ? 0 : N[halo_index_right];
  }

  __syncthreads();

  float Pvalue = 0;
  for(int j = 0; j < Mask_Width; j++) {
    Pvalue += N_ds[threadIdx.x + j]*M[j];
  }
  P[i] = Pvalue;

}
```

Tiled convolution – analysis

- The goal of tiling is to reduce the total number of global memory accesses
- In the original kernel, an internal thread accesses `Mask_Width` elements
  - `blockDim.x`$(2n+1)$ per block
  - 5120 accesses for 1024 threads
- Border threads access fewer elements
  - The missing positions are called *ghost cells*
- How many times a ghost cells accessed depends on its proximity to the array

Tiled convolution – analysis



- The first ghost cell is used by one thread, the second by two threads, etc.
- In general, the number of ghost (i.e. saved) accesses on both sides is $2(1 + 2 + 3 + \ldots + n) = n(n + 1)$
- This becomes insignificant for large arrays and small masks

Tiled convolution – analysis

- For the tiled kernel, each element will be loaded once by one thread
- But we will also load $2n$ halo cells per internal tile
  - n for boundary tiles
- Thus, the total number of memory accesses is $\texttt{blockDim.x}+2n$ for internal tiles
- The ratio of the original vs. tiled kernels is:

$$\frac{\texttt{blockDim.x}(2n-1)}{\texttt{blockDim.x}+2n}$$

- For large arrays and small mask, this can be approximated as:

$$\frac{\texttt{blockDim.x}(2n-1)}{\texttt{blockDim.x}} = 2n+1 = \texttt{Mask\_Width}$$

The ratio is proportional to the mask width