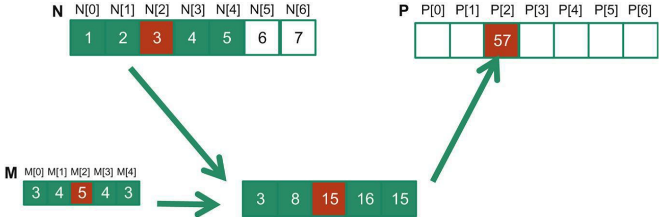
CSc 8530 Parallel Algorithms

Spring 2019

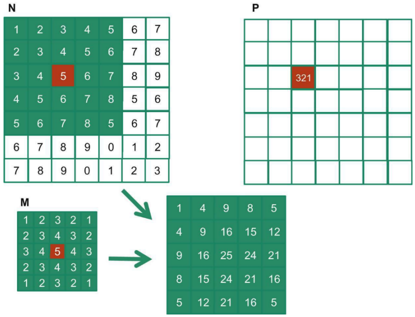
April 23rd, 2019

1D convolution example


N N[0] N[1] N[2] N[3] N[4] N[5] N[6]
1 2 3 4 5 6 7
P P[0] P[1] P[2] P[3] P[4] P[5] P[6]
57
M M[0] M[1] M[2] M[3] M[4]
3 4 5 4 3
3 8 15 16 15

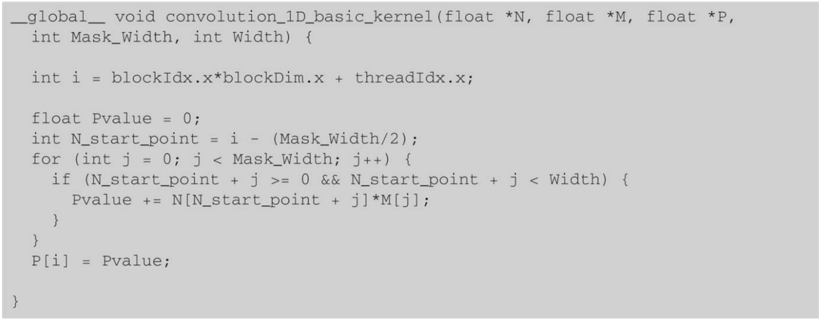
- Here, M and N are the mask and input, resp.
- The value for $N[2]$ is given by the weighted sum of it and its two neighbors on either side
- To calculate $N[3]$ we would slide M over by one to the right
- Typically, masks have odd number of elements so that sum is symmetric around the current element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

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

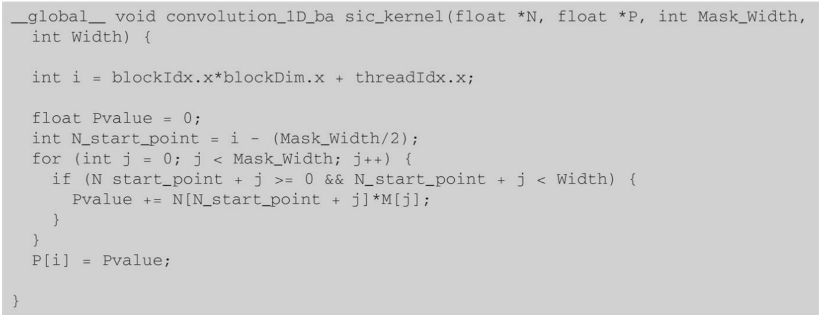
1D convolution: simple CUDA code



```
__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 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

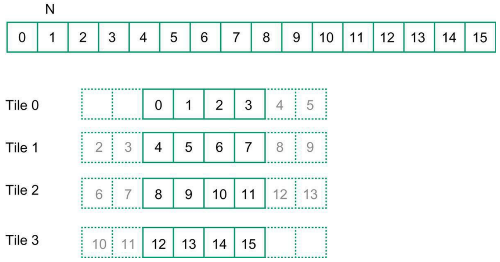
1D convolution: improved CUDA code



```
__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

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 – 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;

}
```

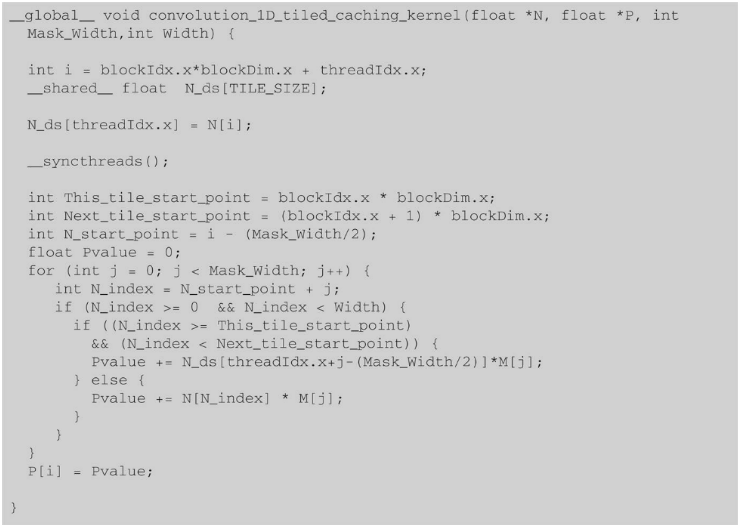
Simpler tiled convolution

- A significant portion of the tiled CUDA code deals with loading halo cells
  - In addition to the internal cells for each tile
- Modern GPUs have L2 and even L3 caches in addition to L1
  - An L1 cache is unique to each SM (hence to each block)
  - L2 and L3 caches are shared among multiple SMs
- Thus, we can simplify our kernel by loading halo cells onto L2 caches
- We take advantage of the fact that halo cells of one tile are internal cells of its neighbors

Simpler tiled convolution

- In our running example, the halo cells of Tile 1, $N[2]$ and $N[3]$ are internal cells of Tile 0
- There is a good probability that by the time Tile 1 accesses these values, they are already in an L2 cache
    - Because Tile 0 had previously requested them
- In this case, we can quickly and transparently access these values multiple times
    - We don't incur additional global memory accesses
- L2 is slower than L1/shared memory
    - But the difference may be negligible for many applications
    - And we avoid having to store these values in the N_{ds} intermediate array

Simplified tiled kernel

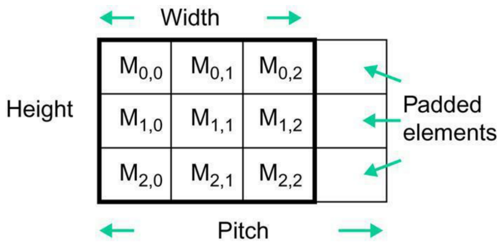


```
__global__ void convolution_1D_tiled_caching_kernel(float *N, float *P, int
  Mask_Width,int Width) {

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

  N_ds[threadIdx.x] = N[i];

  __syncthreads();

  int This_tile_start_point = blockIdx.x * blockDim.x;
  int Next_tile_start_point = (blockIdx.x + 1) * blockDim.x;
  int N_start_point = i - (Mask_Width/2);
  float Pvalue = 0;
  for (int j = 0; j < Mask_Width; j++) {
     int N_index = N_start_point + j;
     if (N_index >= 0  && N_index < Width) {
       if ((N_index >= This_tile_start_point)
         && (N_index < Next_tile_start_point)) {
         Pvalue += N_ds[threadIdx.x+j-(Mask_Width/2)]*M[j];
       } else {
         Pvalue += N[N_index] * M[j];
       }
     }
  }
  P[i] = Pvalue;

}
```

2D convolution with halo cells

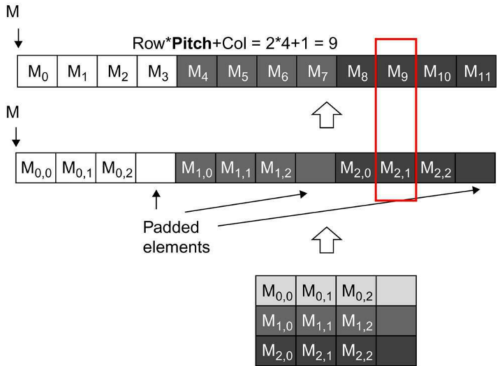
- We will now show how to extend these ideas to 2D convolution
- As before, we will assume that our matrix contains image data
- For added realism, we will assume images are loaded using padding:
  - In practice, nearby items in global memory are best loaded in "bursts" (i.e., several at a time)
  - Images are loaded one row at a time
  - If the row length is not a multiple of the DRAM's burst size (typically a power of two), then the row gets padded with zeros
    - Otherwise, it would take two bursts to load a row
  - This can cause the pointers to the rows to be misaligned w.r.t. the image data

2D convolution with halo cells


Width
Height
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$
Padded elements
Pitch

- Here, a 3×3 image gets padded so that each row has a length of 4
- We can pull each row with a single DRAM burst
- However, we obviously don't want to process these padded elements

Padded matrix access


M
Row*Pitch+Col = 2*4+1 = 9
M_0 M_1 M_2 M_3 M_4 M_5 M_6 M_7 M_8 M_9 M_{10} M_{11}
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$
Padded elements
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$

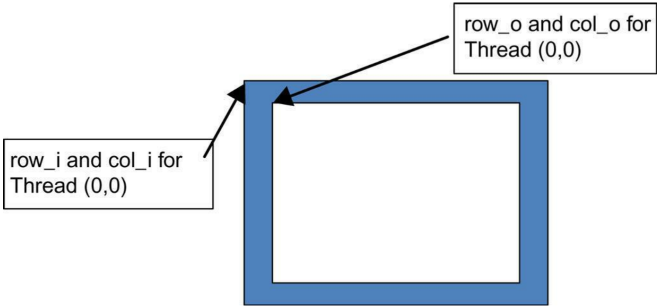
- We have to factor in the pitch size when moving along a linearized array
- The adjusted formula is Row*Pitch + Col

Image format

```
// Image Matrix Structure declaration
//
typedef struct {
   int width;
   int height;
   int pitch;
   int channels;
   float* data;
} * wbImage_t;
```

- Above, we have a typical C struct containing the image data and its properties
  - Similar to Homework 3
- The `channels` variable refers to the number of color channels
  - 1 for grayscale, 3 for RGB

2D ghost cells


row_o and col_o for Thread (0,0)
row_i and col_i for Thread (0,0)

- As with the 1D case, we need to account for ghost cells
- But now along two dimensions

2D tiled kernel code – part 1

```
__global__ void convolution_2D_tiled_kernel(float *P, float *N, int height, int width,
                                            int pitch, int channels, int Mask_Width,
                                            const float __restrict__ *M)
{

  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int row_o = blockIdx.y*O_TILE_WIDTH + ty;
  int col_o = blockIdx.x*O_TILE_WIDTH + tx;

  int row_i = row_o - Mask_Width/2;
  int col_i = col_o - Mask_Width/2;
```

- The `__restrict__` keyword tells the compiler that `M` is read-only
- And thus should be put in the L2 cache
- Different from `__constant__`
  - Constant memory and L2 caches are different parts of the hardware

2D tiled kernel code – part 2

```
__shared__ float N_ds[TILE_SIZE+MAX_MASK_WIDTH-1]
                     [TILE_SIZE+MAX_MASK_HEIGHT-1];
if((row_i >= 0) && (row_i < height) &&
    (col_i >= 0) && (col_i < width)) {
  N_ds[ty][tx] = data[row_i * pitch + col_i];
} else{
  N_ds[ty][tx] = 0.0f;
}
```

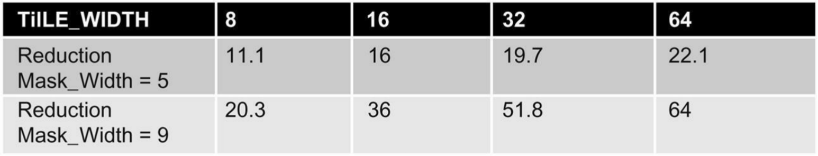
- In this kernel, each block loads *all* the values it needs to compute its output
    - Including ghost and halo cells
- We need to explicitly put zeros for ghost cells
    - Because we won't distinguish between them in the next step
- Also note that this code only works for a single channel

2D tiled kernel code

```
float output = 0.0f;
if(ty < O_TILE_WIDTH && tx < O_TILE_WIDTH){
   for(i = 0; i < MASK_WIDTH; i++) {
     for(j = 0; j < MASK_WIDTH; j++) {
       output += M[i][j] * N_ds[i+ty][j+tx];
     }
   }

   if(row_o < height && col_o < width){
      data[row_o*width + col_o] = output;
   }
 }
```

- We do a nested for-loop to calculate the output values
- Remember that `M` (hopefully) resides in the L2 cache

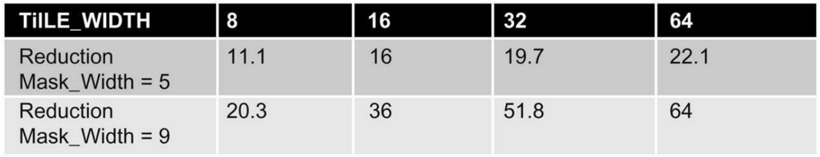
2D tiled kernel code – analysis



TiILE_WIDTH	8	16	32	64
Reduction Mask_Width = 5	11.1	16	19.7	22.1
Reduction Mask_Width = 9	20.3	36	51.8	64

- In a basic kernel, each thread performs $(\text{Mask\_Width})^2$ memory accesses
    - Thus, each block incurs $(\text{Mask\_Width})^2 * (\text{O\_TILE\_WIDTH})^2$ accesses
- In a tiled kernel, each block collective loads one tile
    - So the total memory accesses are $(\text{O\_TILE\_WIDTH}+\text{Mask\_Width-1})^2$

2D tiled kernel code – analysis



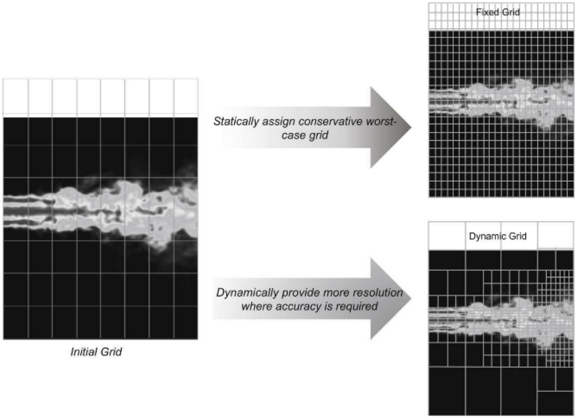
TiILE_WIDTH	8	16	32	64
Reduction Mask_Width = 5	11.1	16	19.7	22.1
Reduction Mask_Width = 9	20.3	36	51.8	64

- Above, we see the ratio as a function of both Mask_Width and O_TILE_WIDTH
- Note that some of the higher values (e.g., Mask_Width = 9 and O_TILE_WIDTH = 64) require more shared memory than is available in current-generation GPUs

CUDA dynamic parallelism

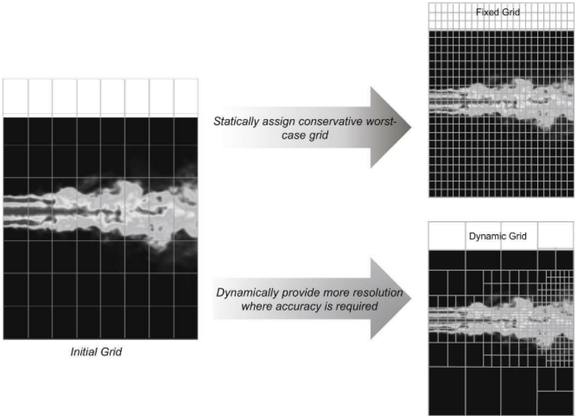
- Up to now, we have assumed that all kernel calls are initiated from the host
- This simplifies the control flow, but makes sophisticated forms of parallelism very hard to implement
    - e.g., recursion, hierarchical grids, time-varying resolution, etc.
- Fortunately, modern CUDA allows kernel functions to initiate additional kernel calls
    - A kernel can even call itself
- This style of programming, **dynamic parallelism**, enables a more flexible and efficient use of computational resources

CUDA dynamic parallelism


Fixed Grid
Statically assign conservative worst-case grid
Dynamic Grid
Dynamically provide more resolution where accuracy is required
Initial Grid

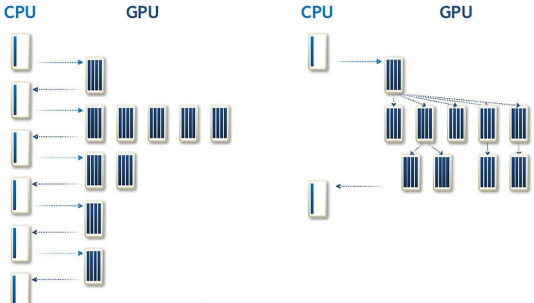
- Data is often spatially, temporally, or structurally heterogenous
- The precision needed to process different subsets may differ

CUDA dynamic parallelism


Fixed Grid
Statically assign conservative worst-case grid
Dynamic Grid
Dynamically provide more resolution where accuracy is required
Initial Grid

- Above, we need more resolution for encoding the simulated air flow than the background
- In standard CUDA, we need to use the maximum necessary resolution across the entire dataset

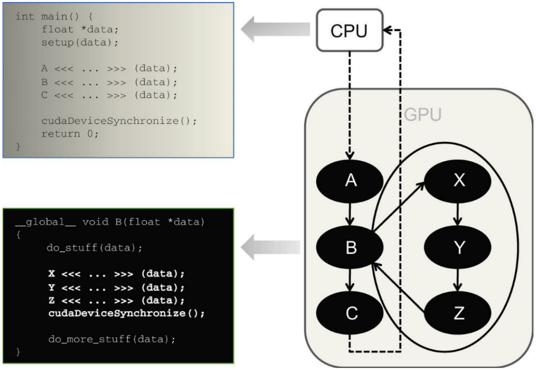
Static vs. dynamic parallelism



(A) *Without Dynamic Parallelism* (B) *With Dynamic Parallelism*

- In static parallelism, the CPU initiates all kernel calls
    - Changes to, e.g., grid resolution, require terminating the current kernel calls
- In dynamic parallelism, we allow GPU kernels to spawn sub-kernels, if needed
    - The GPU's computations are more autonomous from the host

Dynamic parallelism overview



```
int main() {
     float *data;
     setup(data);

     A <<< ... >>> (data);
     B <<< ... >>> (data);
     C <<< ... >>> (data);

     cudaDeviceSynchronize();
     return 0;
}
```

```
__global__ void B(float *data)
{
     do_stuff(data);

     X <<< ... >>> (data);
     Y <<< ... >>> (data);
     Z <<< ... >>> (data);
     cudaDeviceSynchronize();

     do_more_stuff(data);
}
```

- Above, the CPU first launches three GPU kernels A, B, and C
- Kernel B then launches three additional kernels X, Y, and Z
    - This operation would have been invalid in pre-2012 versions of CUDA

Dynamic parallelism overview

- The syntax for launching a kernel from the device is the same as launching it from the host:
- `kernel_name<<<Dg,Db,Ns,S>>>(kernel_arguments])`
- Dg specifies the dimensions and size of the *grid*
- Db specifies the dimensions and size of the *block*
- Ns and S are optimal parameters (we haven't discussed them in class)
  - Ns allows more control over per-thread shared memory allocation
  - S specifies the **stream** associated with the kernel call
    - Streams are used for systems with multiple hosts and/or devices

Static vs. dynamic CUDA parallelism

```
    __global__ void kernel(unsigned  int* start, unsigned int* end,float* someData,
        float* moreData) {

        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
        doSomeWork(someData[i]);

        for(unsigned int j = start[i]; j < end[i]; ++j) {
            doMoreWork(moreData[j]);
        }

    }
```

- We will first analyze a generic program that doesn't do any specific task
  - It still exemplifies important differences between the two styles
  - The above pattern is also common in many applications
    - e.g., `someData` could be the vertices and `moreData` the edges of a graph
- The above code uses **static parallelism**
  - The kernel does not spawn any additional sub-kernels

Static vs. dynamic CUDA parallelism

```
    __global__ void kernel(unsigned  int* start, unsigned int* end,float* someData,
          float* moreData) {

          unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
          doSomeWork(someData[i]);

          for(unsigned int j = start[i]; j < end[i]; ++j) {
                doMoreWork(moreData[j]);
          }

    }
```

- This code has two main limitations:
    1. We cannot parallelize the **for**-loop inside the kernel
    2. If the number of iterations per **for**-loop vary significantly, some threads will end up running for much longer than others
- Both problems are due to insufficient *granularity*
    - Static parallelism only allows one level of granularity or scale

Static vs. dynamic CUDA parallelism

```
    __global__ void kernel_parent(unsigned int* start, unsigned int* end,
        float* someData, float* moreData) {

        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
        doSomeWork(someData[i]);

        kernel_child <<< ceil((end[i]-start[i])/256.0) , 256 >>>
            (start[i], end[i], moreData);

    }

    __global__ void kernel_child(unsigned int start, unsigned int end,
        float* moreData) {

        unsigned int j = start + blockIdx.x*blockDim.x + threadIdx.x;

        if(j < end) {
            doMoreWork(moreData[j]);
        }

    }
```

- Above, we have a dynamically parallel version
- We specify the `kernel_child` function to handle the **for**-loop
- The **if**-statement inside the child kernel acts as the **for**-loop check in the original code

Dynamic parallelism remarks

- The previous example highlighted one of the key design patterns of parallel algorithms:
  - **Replace loops with parallel calls**
  - In principle, every **for**-loop is an opportunity to parallelize your code
  - With dynamic parallelism, we can go down an arbitrary number of nested levels
  - In practice, though, the additional overhead may not be worth the effort for small loops
- In addition, dynamic parallelism erases the software-level distinction between host and device
  - From the point of view of an algorithm, where it gets executed is immaterial