Algorithms:
Threading and Scalability

ECE 6397
Electrical and Computer Engineering
University of Houston
Dr. David Mayerich
Transparent Scalability

- **transparent scalability** – the ability of an algorithm to operate on data of different sizes without reprogramming and user tuning
- maximize the possible number of threads in case the data is large
  - both the block and grid dimensions have maximum values
  - launch each block with as many threads as possible
- Example:
  - launch 101,500 threads with a maximum block dimension of 1024
    - requires between 99 and 100 blocks: \( \frac{101,500}{1024} \approx 99.12 \)
    - if we use 100 blocks, there are too many: \( 100 \times 1024 = 102,400 \)
    - launch more threads, then check for excess threads in the kernel

```c
__global__ void kernelFunc(float* A, float* B, size_t N){
    size_t i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i >= N) return;
    ... //do stuff
}
```
Vector Addition – Configuration

Solve \( C = A + B \) where \( A, B, \) and \( C \in \mathbb{R}^N \)

\[
\begin{bmatrix}
C_1 \\
\vdots \\
C_N
\end{bmatrix} = \begin{bmatrix}
A_1 \\
\vdots \\
A_N
\end{bmatrix} + \begin{bmatrix}
B_1 \\
\vdots \\
B_N
\end{bmatrix}
\]

- \( N \) is unknown at compile time
  - but can be big – which is kind of the point
- Use the maximum number of threads per block
  - allows the largest possible matrix
  - limits to block dimension (\( \approx 1024 \)) and grid dimension (\( \approx 65535 \))

\[
\text{threads} = \text{prop.maxThreadsDim}[0];
\]

- How many blocks should be launched?

\[
\text{blocks} = \text{ceil}((\text{double})N/\text{threads}); \quad // \left\lceil \frac{N}{T} \right\rceil \text{ or round up}
\]

or

\[
\text{blocks} = N/\text{threads} + 1; \quad // \text{sometimes launches extra block}
\]
Vector Addition – Kernel

```c
__global__ void kernelAdd(float* C, float* A, float* B, size_t N) {
    // calculate the index into the array
    size_t i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= N) return; // exit if the thread is hanging
    C[i] = A[i] + B[i];  // perform the addition
}
```

- Not all array sizes will fit perfectly onto a grid
  - some launched threads will not have corresponding array values
  - test for “hanging” threads so that you don’t cause a segmentation fault

- **segmentation fault** – memory access violation, or general protection fault, resulting from an attempted memory access outside of memory allocated to a program
  - the operating system will generally notify you of segfaults on CPU threads
  - this will usually result in a device reset for the GPU
Signal Averaging – Degenerate Cases

• Calculate the time-averaged signal $\tilde{S}$ given the original signal $S$
  • original signal has $N$ samples: $S_i$ where $1 \leq i \leq N$
  • every $T$ samples will be averaged to evaluate $\tilde{S}$
  • how many samples will be in the new signal?
    $\tilde{S}_j$ where $1 \leq j \leq \frac{N}{T}$
  • where can we take advantage of data parallelism?
    map the grid to the output signal

• degenerate cases – exceptional cases where a simple algorithm does not apply
  • what if $N$ is not divisible by $T$?
    • in this case, we will throw excess samples away
    • other options are possible (average remaining samples, extrapolate)
Signal Averaging – Configuration

float* gpu_S; //stores the signal in an array of T values
size_t N; //number of values in the array
size_t T; //number of values averaged
size_t M = N/T; //length of the output signal
float* gpu_Savg; //stores the averaged signal
...

//load all input values, copy to device
int threads = prop.maxThreadsDim[0]; //max # of threads per block
int blocks = ceil((float)M/threads); //launch min number of blocks
kernelAverage<<<blocks, threads>>>(gpu_Savg, gpu_S, M, N);

• The output array gpu_Savg will be M
• I generally prefer adding a default block rather than using the ceil(⋯) function:

  int blocks = M/threads + 1;
  • shorter, easier to understand, doesn’t require re-casting integer values
Signal Averaging – Kernel

```c
__global__ void kernelAverage(float* So, float* Si, size_t M, size_t N) {
    size_t j = blockIdx.x * blockDim.x + threadIdx.x;
    size_t i = j * N;
    if(j >= M) return;
    So[j] = 0;
    for(size_t n = 0; n < N; n++) So[j] += Si[i + n] * 1.0/N;
}
```

• output index j is determined first from the grid configuration
• the input array is exactly N times larger than the output
• `float*` So can also be initialized to zero outside of the kernel:
  ```c
cudaMemset(gpu_Savg, 0, sizeof(float) * M);
```

• I generally prefer initialization inside the kernel, when necessary
• (in this case it isn’t necessary – see next slide)
__global__ void kernelAverage(float* So, float* Si, size_t M, size_t N){
    size_t j = blockIdx.x * blockDim.x + threadIdx.x;
    size_t i = j * N;
    if(j >= M) return;
    So[j] = 0;
    for(size_t n = 0; n < N; n++) So[j] += Si[i + n] * 1.0/N;
}

• register latency ≈ 11 clock cycles
• global memory latency ≈ 300 clock cycles
• replace global fetches with registers whenever possible
  for(size_t n = 0; n < N; n++) So[j] += Si[i + n] * 1.0/N;
  
  float t = 0; //declare a register (much faster, costs a register)
  for(size_t n = 0; n < N; n++) t += Si[i + n] * 1.0/N;
  So[j] = t;
Signal Averaging – Visualization

• Input signal $S$ is time averaged by $N = 4$ to make the output $\bar{S}$
• output array $\bar{S}$:

$$\text{size_t } j = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x};$$

$$j = \begin{bmatrix} [0] & [1] & [M-1] \\ \bar{S}_1 & \bar{S}_2 & \bar{S}_M \end{bmatrix}$$

• input array $S$:

$$\text{size_t } i = j \times N;$$


$Nj \quad Nj + 3 \quad Nj \quad Nj \quad Nj \quad Nj \quad Nj + 1$
Multidimensional Grids

- 1D, 2D, and 3D grids are supported
  - specify grid properties using the \texttt{dim3} structure: \texttt{dim3(x, y, z)}
- common layouts for $M$ blocks composed of $N$ threads:

\begin{verbatim}
\begin{tabular}{|c|c|c|c|}
\hline
\texttt{0} & \texttt{1} & \cdots & \texttt{N-1} \\
\hline
\texttt{b_1} &       &       &       \\
\hline
\vdots     &       &       &       \\
\hline
\texttt{b_{M-1}} &       &       &       \\
\hline
\end{tabular} <<<\texttt{dim3(1,M),N}>>>
\end{verbatim}

\begin{verbatim}
\begin{tabular}{|c|c|c|c|}
\hline
\texttt{0} &       &       &       \\
\hline
\texttt{1} & \texttt{b_1} & \cdots & \texttt{b_{M-1}} \\
\hline
\vdots     & \vdots & \cdots & \vdots \\
\hline
\texttt{N-1} &       &       &       \\
\hline
\end{tabular} <<<\texttt{M,dim3(1,N)}>>>
\end{verbatim}

\begin{verbatim}
\begin{tabular}{|c|c|c|c|}
\hline
\texttt{0} &       & \cdots & \sqrt{N} - 1 \\
\hline
\sqrt{N} - 1 &       & \cdots & \sqrt{N} - 1 \\
\hline
\sqrt{N} - 1 &       & \cdots & \sqrt{N} - 1 \\
\hline
\end{tabular} <<<\texttt{dim3(\sqrt{M},\sqrt{M}),dim3(\sqrt{N},\sqrt{N})}>>>
\end{verbatim}
Matrix Multiplication – Configuration

- calculate $C = AB$ where $A \in \mathbb{R}^{M \times N}$, $B \in \mathbb{R}^{N \times M}$, $C \in \mathbb{R}^{M \times M}$
  - $C_{ij}$ is independent of all other elements:
  $$C_{ij} = \sum_{n=1}^{N} A_{in} \cdot B_{nj}$$

$A =$
\[
\begin{bmatrix}
A_{11} & A_{12} & A_{13} & \cdots & A_{1N} \\
A_{21} & \square & \square & \cdots & \square \\
A_{31} & \square & \square & \cdots & \square \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
A_{M1} & \square & \square & \cdots & \square \\
\end{bmatrix}
\]

$B =$
\[
\begin{bmatrix}
B_{11} & \square & \square & \cdots & \square \\
B_{21} & \square & \square & \cdots & \square \\
B_{31} & \square & \square & \cdots & \square \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
B_{N1} & \square & \square & \cdots & \square \\
\end{bmatrix}
\]

$C =$
\[
\begin{bmatrix}
C_{11} & C_{12} & C_{13} & \cdots & C_{1M} \\
C_{21} & \square & \square & \cdots & \square \\
C_{31} & \square & \square & \cdots & \square \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
C_{M1} & \square & \square & \cdots & \square \\
\end{bmatrix}
\]
Matrix Multiplication – Configuration

```c
float* gpu_A; //pointer to A, \( A_{ij} = A[i*M+j] \)
float* gpu_B; //pointer to B, \( B_{ij} = B[i*N+j] \)
float* gpu_C; //pointer to C, \( C_{ij} = C[i*M+j] \)
size_t M, N; //matrix sizes
...
//load data and copy to the GPU
```

```
dim3 threads(sqrt(props.maxThreadsPerBlock), sqrt(props.maxThreadsPerBlock));
dim3 blocks(M/threads.x+1, M/threads.y+1);
kernelMatrixMult<<<blocks, threads>>>(gpu_C, gpu_A, gpu_B, M, N)
```

- this configuration will create square blocks that are \( \sqrt{T_{\text{max}}} \times \sqrt{T_{\text{max}}} \)
- other options are viable:

```c
int threads = prop.maxThreadsDim[0];
dim3 blocks(M/threads+1, M);
```

- this configuration is compatible with any set of positive integer values \( M \) and \( N \)
**Matrix Multiplication – Kernel**

```
__global__ void kernelMatrixMult(float* C, float* A, float* B, size_t M, size_t N)
{
    size_t i = blockIdx.y * blockDim.y + threadIdx.y;   //calculate the i (row) index
    size_t j = blockIdx.x * blockDim.x + threadIdx.x;   //calculate the j (column) index
    if(i >= M || j >= M) return; //return if (i,j) is outside the matrix
    float c = 0; //initialize a register to store the result
    for(size_t n = 0; n < N; n++) //for each element in the dot product
        c += A[n*M+i] * B[j*N+n]; //perform a multiply-add
    C[i*M + j] = c; //send the register value to global memory
}
```

- note that all proposed configurations will work with this kernel
- both indices must be checked against the maximum dimension
- what is the most important line in this kernel? Is there any way to make it more efficient?

  ```
  A += i; //move the A pointer to the start of the column
  size_t jN = j * N; //pre-multiply j*N
  for(size_t n = 0; n < N; n++) c+= A[n*M] * B[jN + n];
  ```