Introduction to High Performance Computing

\\

SDS406 – Fall semester, 2024 - 2025

\\

L06: GPU programming, 4th November 2024

Outline

Last week

- Review of GPU architecture
- Review of GPU programming and CUDA
- Some details of the GPU nodes of Cyclamen

Outline

Last week

- Review of GPU architecture
- Review of GPU programming and CUDA
- Some details of the GPU nodes of Cyclamen

Practical examples on GPUs

Covering:

- GPU performance vs CPU performance — ✓
- Memory coalescing on GPUs — ✓
- Shared memory — this week
- Details of GPU thread scheduling (warps) and why you should care — this week

CUDA, another example

Exercise: rotate and shift an array of (x, y) coordinates

- `/onyx/data/sds406f24/l06/ex01/rot.cu` calls, as before, the same kernel twice
- Operation is $\vec{v}_i = U\vec{r}_i + \vec{s}_i$
- Where:

$$U = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$$

CUDA, another example

Exercise: rotate and shift an array of (x, y) coordinates

- `/onyx/data/sds406f24/l06/ex01/rot.cu` calls, as before, the same kernel twice
- Operation is $\vec{v}_i = U\vec{r}_i + \vec{s}_i$
- Where:

$$U = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$$

- Equivalently:

$$v_{i,x} = \cos(\theta) r_{i,x} - \sin(\theta) r_{i,y} + s_{i,x}$$

$$v_{i,y} = \sin(\theta) r_{i,x} + \cos(\theta) r_{i,y} + s_{i,y}$$

Coordinate transformation using CUDA

- $\vec{v}_i = U\vec{r}_i + \vec{s}_i \Rightarrow$

$$v_{i,x} = cos(\theta)r_{i,x} - sin(\theta)r_{i,y} + s_{i,x}$$

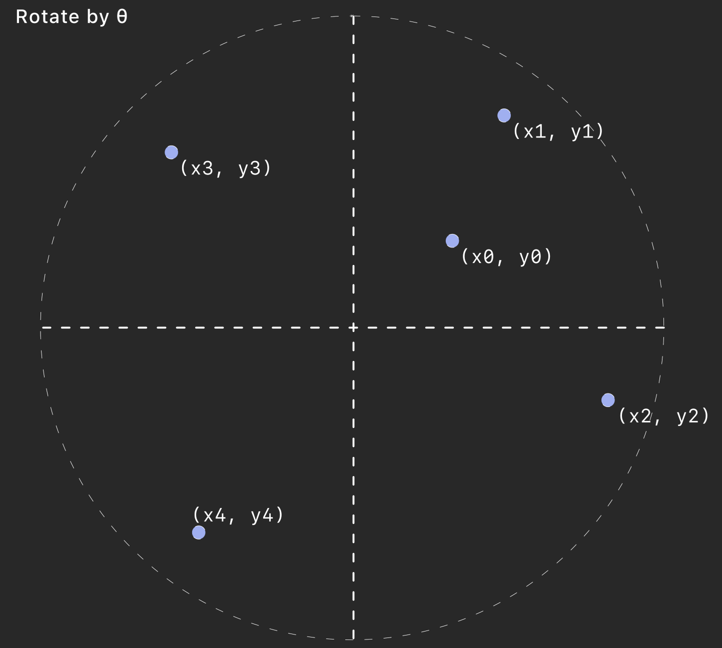
$$v_{i,y} = sin(\theta)r_{i,x} + cos(\theta)r_{i,y} + s_{i,y}$$

Coordinate transformation using CUDA

- $\vec{v}_i = U\vec{r}_i + \vec{s}_i \Rightarrow$

$$v_{i,x} = \cos(\theta) r_{i,x} - \sin(\theta) r_{i,y} + s_{i,x}$$

$$v_{i,y} = \sin(\theta) r_{i,x} + \cos(\theta) r_{i,y} + s_{i,y}$$

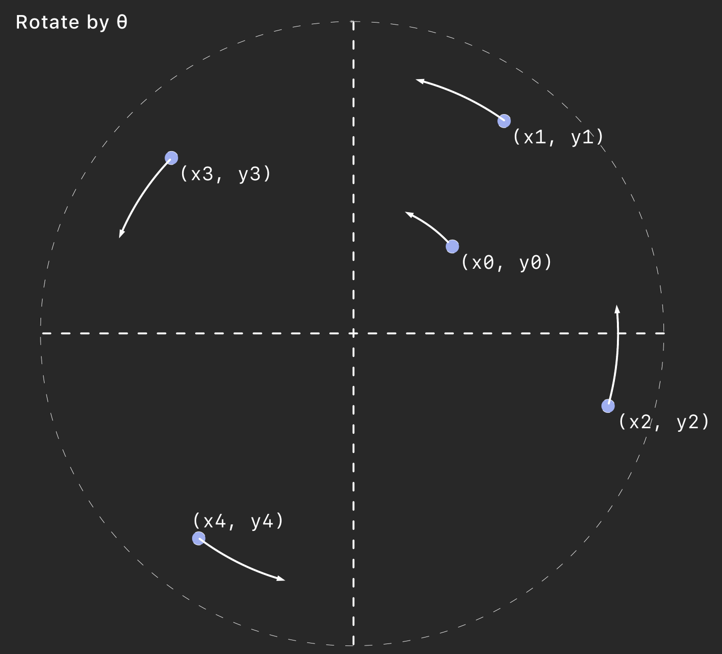


Coordinate transformation using CUDA

- $\vec{v}_i = U\vec{r}_i + \vec{s}_i \Rightarrow$

$$v_{i,x} = \cos(\theta) r_{i,x} - \sin(\theta) r_{i,y} + s_{i,x}$$

$$v_{i,y} = \sin(\theta) r_{i,x} + \cos(\theta) r_{i,y} + s_{i,y}$$

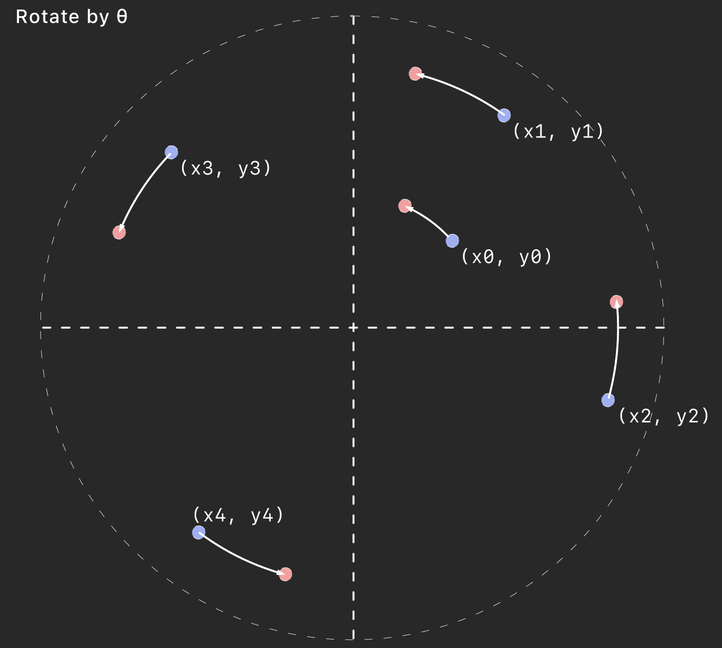


Coordinate transformation using CUDA

- $\vec{v}_i = U\vec{r}_i + \vec{s}_i \Rightarrow$

$$v_{i,x} = \cos(\theta) r_{i,x} - \sin(\theta) r_{i,y} + s_{i,x}$$

$$v_{i,y} = \sin(\theta) r_{i,x} + \cos(\theta) r_{i,y} + s_{i,y}$$

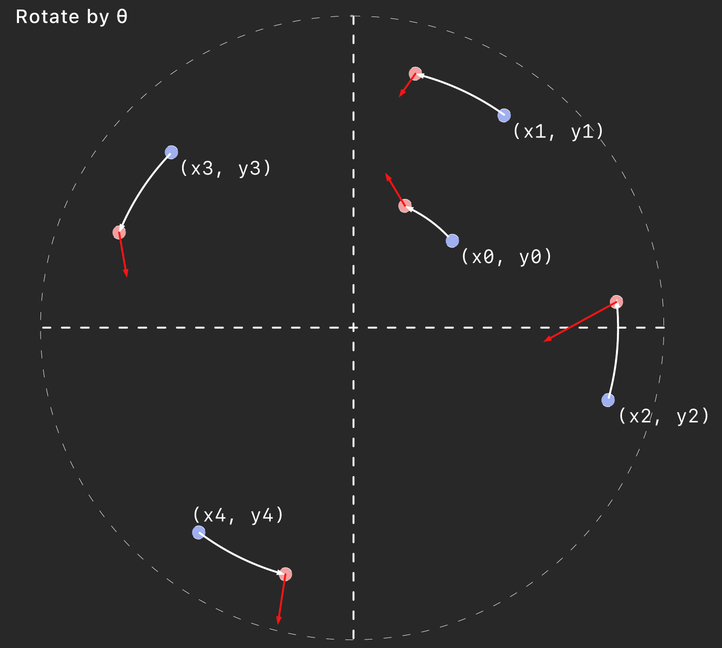


Coordinate transformation using CUDA

- $\vec{v}_i = U\vec{r}_i + \vec{s}_i \Rightarrow$

$$v_{i,x} = \cos(\theta) r_{i,x} - \sin(\theta) r_{i,y} + s_{i,x}$$

$$v_{i,y} = \sin(\theta) r_{i,x} + \cos(\theta) r_{i,y} + s_{i,y}$$

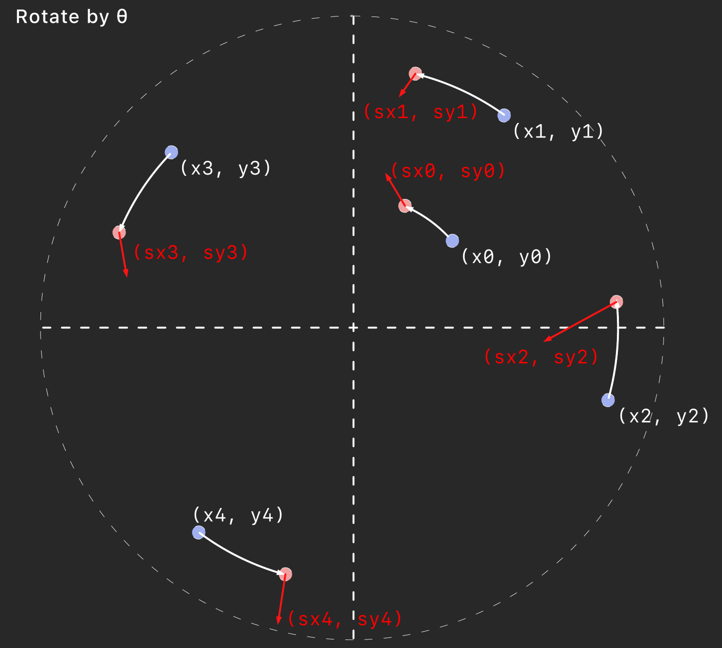


Coordinate transformation using CUDA

- $\vec{v}_i = U\vec{r}_i + \vec{s}_i \Rightarrow$

$$v_{i,x} = \cos(\theta)r_{i,x} - \sin(\theta)r_{i,y} + s_{i,x}$$

$$v_{i,y} = \sin(\theta)r_{i,x} + \cos(\theta)r_{i,y} + s_{i,y}$$

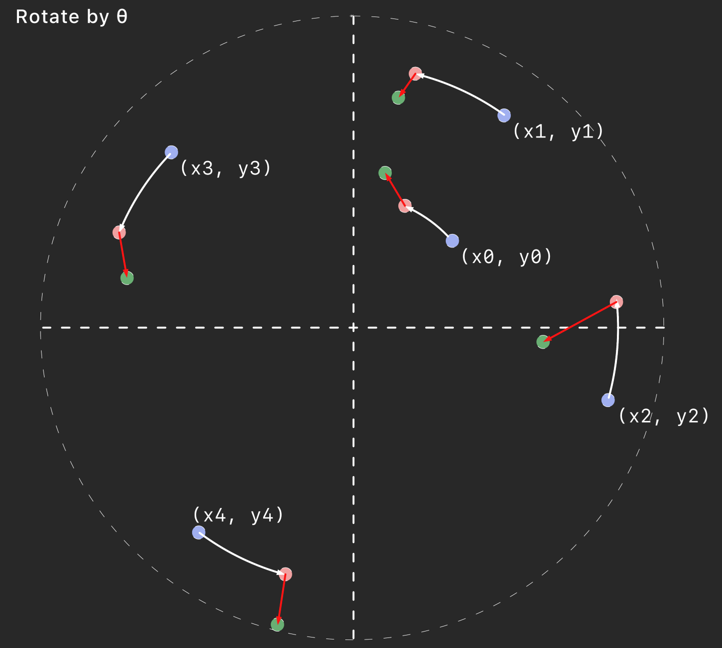


Coordinate transformation using CUDA

- $\vec{v}_i = U\vec{r}_i + \vec{s}_i \Rightarrow$

$$v_{i,x} = \cos(\theta)r_{i,x} - \sin(\theta)r_{i,y} + s_{i,x}$$

$$v_{i,y} = \sin(\theta)r_{i,x} + \cos(\theta)r_{i,y} + s_{i,y}$$

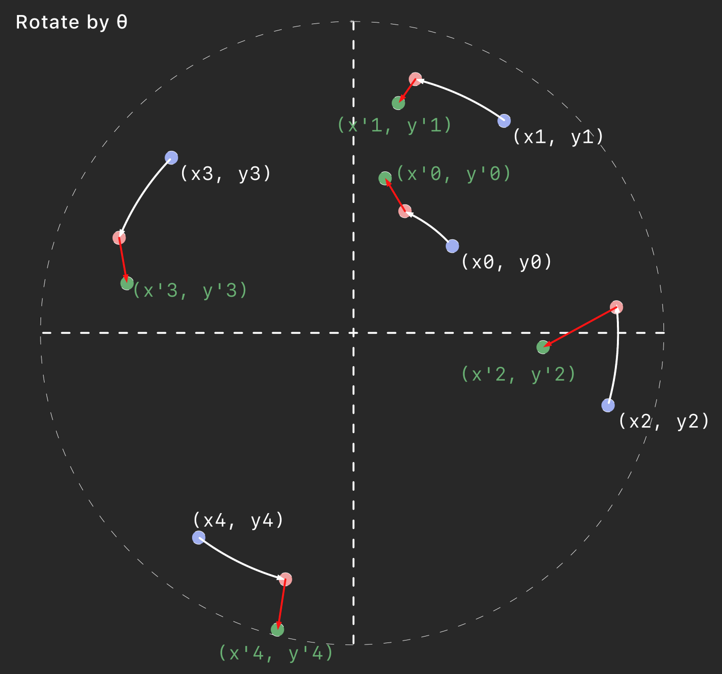


Coordinate transformation using CUDA

- $\vec{v}_i = U\vec{r}_i + \vec{s}_i \Rightarrow$

$$v_{i,x} = \cos(\theta) r_{i,x} - \sin(\theta) r_{i,y} + s_{i,x}$$

$$v_{i,y} = \sin(\theta) r_{i,x} + \cos(\theta) r_{i,y} + s_{i,y}$$



Coordinate transformation using CUDA

TODO, for a first version

- Implement a CUDA version for the second call
- Each GPU thread operating on one point (i)

Coordinate transformation using CUDA

TODO, for a first version

- Implement a CUDA version for the second call
- Each GPU thread operating on one point (i)

Example:

```
[ikoutsou@front02 ex01]$ export OMP_PROC_BIND="close"
[ikoutsou@front02 ex01]$ export OMP_PLACES="cores"
[ikoutsou@front02 ex01]$ export OMP_NUM_THREADS=16
[ikoutsou@front02 ex01]$ nvcc -arch=sm_60 -O3 -Xcompiler -fopenmp -o rot rot.cu
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=16 -p p100  --gres=gpu:1 ./rot 32 $((1024*1024*128))
 CPU: nthr =    16   t0 = 0.0806 sec   P =  13.329 Gflop/s   B =  39.988 GB/s
 GPU: nthr =    32   t0 = 0.0076 sec   P = 141.077 Gflop/s   B = 423.231 GB/s
 Diff = 1.115821e-15
```

Coordinate transformation using CUDA

The optimal number of threads typically needs to be obtained empirically

- If we allow the number of threads to be a command line argument, we can easily scan for it

```
[ikoutsou@front02 ex01]$ for((th=4; th<=1024; th*=2))
> do srun -n 1 --cpus-per-task=16 -p p100  --gres=gpu:1 ./rot $th $((1024*1024*128))
> done 2>&1 | grep GPU
 GPU: nthr =    4   t0 = 0.0630 sec   P =  17.047 Gflop/s   B =  51.142 GB/s
 GPU: nthr =    8   t0 = 0.0313 sec   P =  34.341 Gflop/s   B = 103.023 GB/s
 GPU: nthr =   16   t0 = 0.0149 sec   P =  71.832 Gflop/s   B = 215.497 GB/s
 GPU: nthr =   32   t0 = 0.0076 sec   P = 141.077 Gflop/s   B = 423.231 GB/s
 GPU: nthr =   64   t0 = 0.0065 sec   P = 166.240 Gflop/s   B = 498.719 GB/s
 GPU: nthr =  128   t0 = 0.0065 sec   P = 165.598 Gflop/s   B = 496.794 GB/s
 GPU: nthr =  256   t0 = 0.0065 sec   P = 165.933 Gflop/s   B = 497.800 GB/s
 GPU: nthr =  512   t0 = 0.0064 sec   P = 167.277 Gflop/s   B = 501.831 GB/s
 GPU: nthr = 1024   t0 = 0.0064 sec   P = 168.327 Gflop/s   B = 504.982 GB/s
```

Coordinate transformation using CUDA

The optimal number of threads typically needs to be obtained empirically

- If we allow the number of threads to be a command line argument, we can easily scan for it

```
[ikoutsou@front02 ex01]$ for((th=4; th⩽1024; th*=2))
> do srun -n 1 --cpus-per-task=16 -p p100  --gres=gpu:1 ./rot $th $((1024*1024*128))
> done 2>&1 | grep GPU
 GPU: nthr =    4   t0 = 0.0630 sec   P =  17.047 Gflop/s   B =  51.142 GB/s
 GPU: nthr =    8   t0 = 0.0313 sec   P =  34.341 Gflop/s   B = 103.023 GB/s
 GPU: nthr =   16   t0 = 0.0149 sec   P =  71.832 Gflop/s   B = 215.497 GB/s
 GPU: nthr =   32   t0 = 0.0076 sec   P = 141.077 Gflop/s   B = 423.231 GB/s
 GPU: nthr =   64   t0 = 0.0065 sec   P = 166.240 Gflop/s   B = 498.719 GB/s
 GPU: nthr =  128   t0 = 0.0065 sec   P = 165.598 Gflop/s   B = 496.794 GB/s
 GPU: nthr =  256   t0 = 0.0065 sec   P = 165.933 Gflop/s   B = 497.800 GB/s
 GPU: nthr =  512   t0 = 0.0064 sec   P = 167.277 Gflop/s   B = 501.831 GB/s
 GPU: nthr = 1024   t0 = 0.0064 sec   P = 168.327 Gflop/s   B = 504.982 GB/s
```

- Tops at ~500 GBytes/s or ~70%. Can we do better?

Coordinate transformation using CUDA

Optimizations

- Note the loading of elements of `r[]` from global memory $\Rightarrow$ two continuous component per thread
- Optimization opportunity: use one thread per *component*

Coordinate transformation using CUDA

Optimizations

- Note the loading of elements of `r[]` from global memory $\Rightarrow$ two continuous component per thread
- Optimization opportunity: use one thread per *component*

  $\Rightarrow$ Instead of each thread operating on one point, consider each thread operating on a single component of the coordinate

Coordinate transformation using CUDA

Optimizations

- Note the loading of elements of `r[]` from global memory $\Rightarrow$ two continuous component per thread
- Optimization opportunity: use one thread per *component*

  $\Rightarrow$ Instead of each thread operating on one point, consider each thread operating on a single component of the coordinate
- In other words, have:
  - even threads computing the `x` coordinate part of `v[:]`
  - odd threads computing the `y` coordinate of `v[:]`

Coordinate transformation using CUDA

Optimizations

- Note the loading of elements of `r[]` from global memory $\Rightarrow$ two continuous component per thread
- Optimization opportunity: use one thread per *component*

  $\Rightarrow$ Instead of each thread operating on one point, consider each thread operating on a single component of the coordinate
- In other words, have:
  - even threads computing the `x` coordinate part of `v[:]`
  - odd threads computing the `y` coordinate of `v[:]`
- This will demonstrate the use of *shared memory*, i.e. fast memory which all threads in a single block can access

Coordinate transformation using CUDA

Optimizations

- Note the loading of elements of `r[]` from global memory $\Rightarrow$ two continuous component per thread
- Optimization opportunity: use one thread per *component*

  $\Rightarrow$ Instead of each thread operating on one point, consider each thread operating on a single component of the coordinate
- In other words, have:
  - even threads computing the `x` coordinate part of `v[:]`
  - odd threads computing the `y` coordinate of `v[:]`
- This will demonstrate the use of *shared memory*, i.e. fast memory which all threads in a single block can access
- Shared memory is declared with the shared attribute, i.e.:

  ```
  __shared__ float arr[SIZE];
  ```

Coordinate transformation using CUDA

Optimizations

- Note the loading of elements of `r[]` from global memory $\Rightarrow$ two continuous component per thread
- Optimization opportunity: use one thread per *component*

  $\Rightarrow$ Instead of each thread operating on one point, consider each thread operating on a single component of the coordinate
- In other words, have:
  - even threads computing the `x` coordinate part of `v[:]`
  - odd threads computing the `y` coordinate of `v[:]`
- This will demonstrate the use of *shared memory*, i.e. fast memory which all threads in a single block can access
- Shared memory is declared with the shared attribute, i.e.:

```
__shared__ float arr[SIZE];
```

- Note that here `SIZE` **must** be known at **compile time**

Coordinate transformation using CUDA

Optimizations

- Note the loading of elements of `r[]` from global memory $\Rightarrow$ two continuous component per thread
- Optimization opportunity: use one thread per *component*

  $\Rightarrow$ Instead of each thread operating on one point, consider each thread operating on a single component of the coordinate
- In other words, have:
  - even threads computing the `x` coordinate part of `v[:]`
  - odd threads computing the `y` coordinate of `v[:]`
- This will demonstrate the use of *shared memory*, i.e. fast memory which all threads in a single block can access
- Shared memory is declared with the shared attribute, i.e.:

```
__shared__ float arr[SIZE];
```

- Note that here `SIZE` **must** be known at **compile time**
- Alternatively, we can have dynamic allocation of shared memory (relatively recent CUDA feature)

Coordinate transformation using CUDA

Optimizations

- Below is how we would like to organize this calculation:

```
i2=2*i
(x coord. of elem. i + 0) thread = 0; v[i2+0] = r[i2+0]*ct - r[i2+1]*st + s[i2+0]
(y coord. of elem. i + 0) thread = 1; v[i2+1] = r[i2+1]*ct + r[i2+0]*st + s[i2+1]
(x coord. of elem. i + 1) thread = 2; v[i2+2] = r[i2+2]*ct - r[i2+3]*st + s[i2+2]
(y coord. of elem. i + 1) thread = 3; v[i2+3] = r[i2+3]*ct + r[i2+2]*st + s[i2+3]
(x coord. of elem. i + 2) thread = 4; v[i2+4] = r[i2+4]*ct - r[i2+5]*st + s[i2+4]
(y coord. of elem. i + 2) thread = 5; v[i2+5] = r[i2+5]*ct + r[i2+4]*st + s[i2+5]
(x coord. of elem. i + 3) thread = 6; v[i2+6] = r[i2+6]*ct - r[i2+7]*st + s[i2+6]
(y coord. of elem. i + 3) thread = 7; v[i2+7] = r[i2+7]*ct + r[i2+6]*st + s[i2+7]
...
```

- Notice that odd threads and even threads carry out different operations
- But on a GPU, it is important for performance to have all threads in a kernel execute the **same** operations
- In other words, try to avoid as much as possible constructs like:
  `if(ithr % 2 == 0){ ... };`

Coordinate transformation using CUDA

Optimizations

- First define a macro at the beginning of the file:

```
#define MAX_THR 1024
```

- Then, when invoking the kernel, change the call to use twice the number of blocks:

```
gpu_rotate<<<2*n/n_gpu_thr, n_gpu_thr>>>(n, d_v, theta, d_r, d_s);
```

Coordinate transformation using CUDA

Optimizations

- In the kernel, declare a shared array, to be used to store the elements of `r[]`:

```
__shared__ float rr[MAX_THR];
```

Coordinate transformation using CUDA

Optimizations

- In the kernel, declare a shared array, to be used to store the elements of `r[]`:

```
__shared__ float rr[MAX_THR];
```

- We need a shared array for `r[]`, because different threads will need to access the same elements. In particular, whether odd or even, each thread needs to access both `x` and `y` components of `x[]`
- By reading `r[]` into `rr[]` once, we avoid each thread having to read elements of `r[]` twice from global memory, which is slow

Coordinate transformation using CUDA

Optimizations

- In the kernel, declare a shared array, to be used to store the elements of `r[]`:

```
__shared__ float rr[MAX_THR];
```

- We need a shared array for `r[]`, because different threads will need to access the same elements. In particular, whether odd or even, each thread needs to access both `x` and `y` components of `x[]`
- By reading `r[]` into `rr[]` once, we avoid each thread having to read elements of `r[]` twice from global memory, which is slow
- Read the elements of `r[]` corresponding to this block into `rr[]`:

```
  int idx = iblk*nthr + ithr;
  rr[ithr] = r[idx];
```

Coordinate transformation using CUDA

Optimizations

- In the kernel, declare a shared array, to be used to store the elements of `r[]`:

```
__shared__ float rr[MAX_THR];
```

- We need a shared array for `r[]`, because different threads will need to access the same elements. In particular, whether odd or even, each thread needs to access both `x` and `y` components of `x[]`
- By reading `r[]` into `rr[]` once, we avoid each thread having to read elements of `r[]` twice from global memory, which is slow
- Read the elements of `r[]` corresponding to this block into `rr[]`:

```
  int idx = iblk*nthr + ithr;
  rr[ithr] = r[idx];
```

This way, the loading is done parallel: each thread reads in one component of `r[]`

Coordinate transformation using CUDA

Optimizations

- Now insert the following, which only achieves the operation partially:

```
float rs = s[idx] + ct*rr[ithr];
```

Coordinate transformation using CUDA

Optimizations

- Now insert the following, which only achieves the operation partially:

```
float rs = s[idx] + ct*rr[ithr];
```

- The operation is still incomplete; what we have achieved with the above is:

$$v_x \leftarrow \cos(\theta) r_x + s_x$$

$$v_y \leftarrow \cos(\theta) r_y + s_y$$

Coordinate transformation using CUDA

Optimizations

- Now insert the following, which only achieves the operation partially:

```
float rs = s[idx] + ct*rr[ithr];
```

- The operation is still incomplete; what we have achieved with the above is:

$$v_x \leftarrow cos(\theta)r_x + s_x$$

$$v_y \leftarrow cos(\theta)r_y + s_y$$

we are missing:

$$v_x \leftarrow v_x - sin(\theta)r_y$$

$$v_y \leftarrow v_y + sin(\theta)r_x$$

Coordinate transformation using CUDA

Optimizations

- We are missing:

$$v_x \leftarrow v_x - sin(\theta)r_y$$

$$v_y \leftarrow v_y + sin(\theta)r_x$$

Coordinate transformation using CUDA

Optimizations

- We are missing:

$$v_x \leftarrow v_x - sin(\theta) r_y$$

$$v_y \leftarrow v_y + sin(\theta) r_x$$

- Consider the following:

```
int sw = 1 - 2*(ithr & 1);
```

Coordinate transformation using CUDA

Optimizations

- We are missing:

$$v_x \leftarrow v_x - sin(\theta)r_y$$

$$v_y \leftarrow v_y + sin(\theta)r_x$$

- Consider the following:

```
int sw = 1 - 2*(ithr & 1);
```

- `&` is a bitwise "and" operation, meaning `ithr & 1` will evaluate to:
- `0` if `ithr` is even
- `1` if `ithr` is odd

Coordinate transformation using CUDA

Optimizations

- We are missing:

$$v_x \leftarrow v_x - sin(\theta)r_y$$

$$v_y \leftarrow v_y + sin(\theta)r_x$$

- Consider the following:

```
int sw = 1 - 2*(ithr & 1);
```

- `&` is a bitwise "and" operation, meaning `ithr & 1` will evaluate to:
- `0` if `ithr` is even
- `1` if `ithr` is odd

`sw = 1 - 2*(ithr & 1)` therefore yields:

```
ithr = 0,  1, 2,  3,  ...
sw   = 1, -1, 1, -1,  ...
```

Coordinate transformation using CUDA

Optimizations

- We are missing:

$$v_x \leftarrow v_x - sin(\theta) r_y$$

$$v_y \leftarrow v_y + sin(\theta) r_x$$

Coordinate transformation using CUDA

Optimizations

- We are missing:

$$v_x \leftarrow v_x - sin(\theta) r_y$$

$$v_y \leftarrow v_y + sin(\theta) r_x$$

- Consider:

```
rs =  rs - sw*st*rr[ithr+sw];
```

Coordinate transformation using CUDA

Optimizations

- We are missing:

$$v_x \leftarrow v_x - sin(\theta) r_y$$

$$v_y \leftarrow v_y + sin(\theta) r_x$$

- Consider:

```
rs =  rs - sw*st*rr[ithr+sw];
```

- Then read back into `out[]`:

```
  out[idx] = rs;
```

Coordinate transformation using CUDA

Optimizations

- Compile and run, scanning the number of GPU threads (filtering only the GPU line):

```
[ikoutsou@front02 ex01]$ for((th=4; th⩽1024; th*=2))
> do srun -n 1 --cpus-per-task=8 -p nehalem  --gres=gpu:1 ./rot $th $((1024*1024*128))
> done 2>&1 | grep GPU
 GPU: nthr =    4   t0 = 0.1217 sec   P =   8.822 Gflop/s   B =  26.465 GB/s
 GPU: nthr =    8   t0 = 0.0608 sec   P =  17.664 Gflop/s   B =  52.993 GB/s
 GPU: nthr =   16   t0 = 0.0283 sec   P =  37.941 Gflop/s   B = 113.824 GB/s
 GPU: nthr =   32   t0 = 0.0155 sec   P =  69.153 Gflop/s   B = 207.459 GB/s
 GPU: nthr =   64   t0 = 0.0078 sec   P = 137.150 Gflop/s   B = 411.450 GB/s
 GPU: nthr =  128   t0 = 0.0059 sec   P = 180.614 Gflop/s   B = 541.841 GB/s
 GPU: nthr =  256   t0 = 0.0060 sec   P = 180.252 Gflop/s   B = 540.756 GB/s
 GPU: nthr =  512   t0 = 0.0060 sec   P = 179.433 Gflop/s   B = 538.300 GB/s
 GPU: nthr = 1024   t0 = 0.0061 sec   P = 175.278 Gflop/s   B = 525.835 GB/s
```

Coordinate transformation using CUDA

Optimizations

- Compile and run, scanning the number of GPU threads (filtering only the GPU line):

```
[ikoutsou@front02 ex01]$ for((th=4; th≤1024; th*=2))
> do srun -n 1 --cpus-per-task=8 -p nehalem  --gres=gpu:1 ./rot $th $((1024*1024*128))
> done 2>&1 | grep GPU
 GPU: nthr =    4   t0 = 0.1217 sec   P =   8.822 Gflop/s   B =  26.465 GB/s
 GPU: nthr =    8   t0 = 0.0608 sec   P =  17.664 Gflop/s   B =  52.993 GB/s
 GPU: nthr =   16   t0 = 0.0283 sec   P =  37.941 Gflop/s   B = 113.824 GB/s
 GPU: nthr =   32   t0 = 0.0155 sec   P =  69.153 Gflop/s   B = 207.459 GB/s
 GPU: nthr =   64   t0 = 0.0078 sec   P = 137.150 Gflop/s   B = 411.450 GB/s
 GPU: nthr =  128   t0 = 0.0059 sec   P = 180.614 Gflop/s   B = 541.841 GB/s
 GPU: nthr =  256   t0 = 0.0060 sec   P = 180.252 Gflop/s   B = 540.756 GB/s
 GPU: nthr =  512   t0 = 0.0060 sec   P = 179.433 Gflop/s   B = 538.300 GB/s
 GPU: nthr = 1024   t0 = 0.0061 sec   P = 175.278 Gflop/s   B = 525.835 GB/s
```

- Maximum performance saturates at ~540 GB/s, or ~75% of peak bandwidth

Matrix-vector multiplication

We will look into another example, the matrix vector multiplication

$$y = Ax$$

where y, x are vectors (1-dimensional) and A is a matrix (2-dimensional)

- In the general case A is not square
- $A_{M\times N}, x_N, y_M$

Matrix-vector multiplication

We will look into another example, the matrix vector multiplication

$$y = Ax$$

where y, x are vectors (1-dimensional) and A is a matrix (2-dimensional)

- In the general case A is not square
- $A_{M\times N}, x_N, y_M$

$$y_i = \sum_{j=0}^{N-1} A_{ij} x_j, \quad i = 0, \ldots, M$$

Matrix-vector multiplication

We will look into another example, the matrix vector multiplication

$$y = Ax$$

where y, x are vectors (1-dimensional) and A is a matrix (2-dimensional)

- In the general case A is not square
- $A_{M \times N}, x_N, y_M$

$$y_i = \sum_{j=0}^{N-1} A_{ij} x_j, \;\; i = 0, \ldots, M$$

```
for(int i=0; i<m; i++) {
  y[i] = 0;
  for(int j=0; j<n; j++) {
    y[i] = y[i] + A[i][j] * x[j];
  }
}
```

Matrix-vector multiplication

We will look into another example, the matrix vector multiplication

$$y = Ax$$

where y, x are vectors (1-dimensional) and A is a matrix (2-dimensional)

- In the general case A is not square
- $A_{M\times N}, x_N, y_M$

$$y_i = \sum_{j=0}^{N-1} A_{ij} x_j, \;\; i = 0, \ldots, M$$

```
for(int i=0; i<m; i++) {
  y[i] = 0;
  for(int j=0; j<n; j++) {
    y[i] = y[i] + A[i][j] * x[j];
  }
}
```

```
for(int i=0; i<m; i++) {
  y[i] = 0;
  for(int j=0; j<n; j++) {
    y[i] += A[i*n + j] * x[j];
  }
}
```

Matrix-vector multiplication

Take `/onyx/data/sds406f24/l06/ex02/ex02/.` for the CPU code:

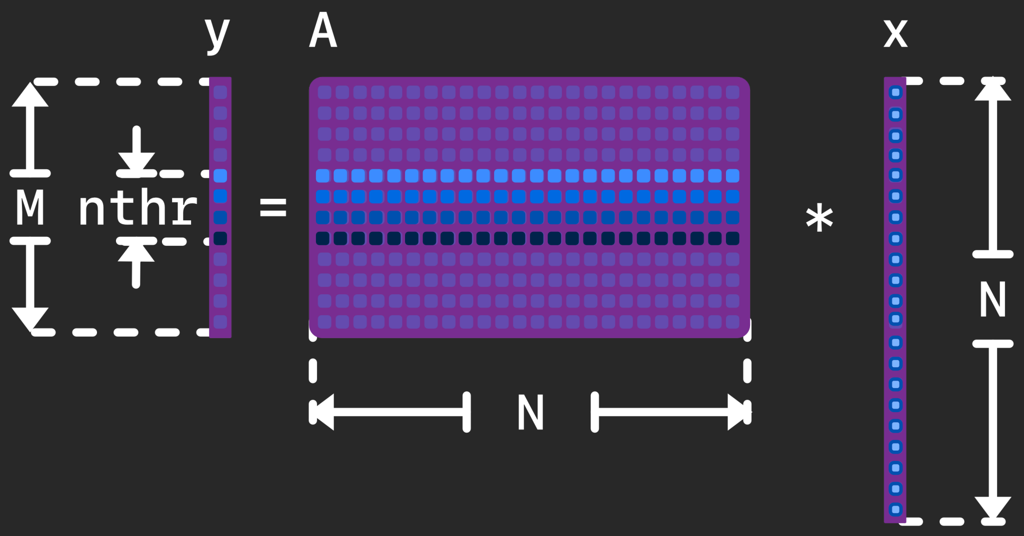
```
[ikoutsou@front02 l06]$ cp -r /onyx/data/sds406f24/l06/ex02/ex02 .
[ikoutsou@front02 l06]$ cd ex02/.
[ikoutsou@front02 ex02]$ nvcc -arch=sm_60 -O3 -Xcompiler -fopenmp -o matvec matvec.cu
[ikoutsou@front02 ex02]$ export OMP_PLACES="cores"
[ikoutsou@front02 ex02]$ export OMP_PROC_BIND="close"
[ikoutsou@front02 ex02]$ export OMP_NUM_THREADS=16
[ikoutsou@front02 ex02]$ srun -N 1  --cpus-per-task=16 -p p100  --gres=gpu:1 ./matvec 4096 8192
 CPU: nthr =   16   t0 = 0.0036 sec   P =  18.888 Gflop/s   B =  37.791 GB/s
 CPU: nthr =   16   t0 = 0.0030 sec   P =  22.663 Gflop/s   B =  45.343 GB/s
 Diff = 0.000000e+00
```

Matrix-vector multiplication

Our task is to modify the second call of the `Ax()` function to run on the GPU.

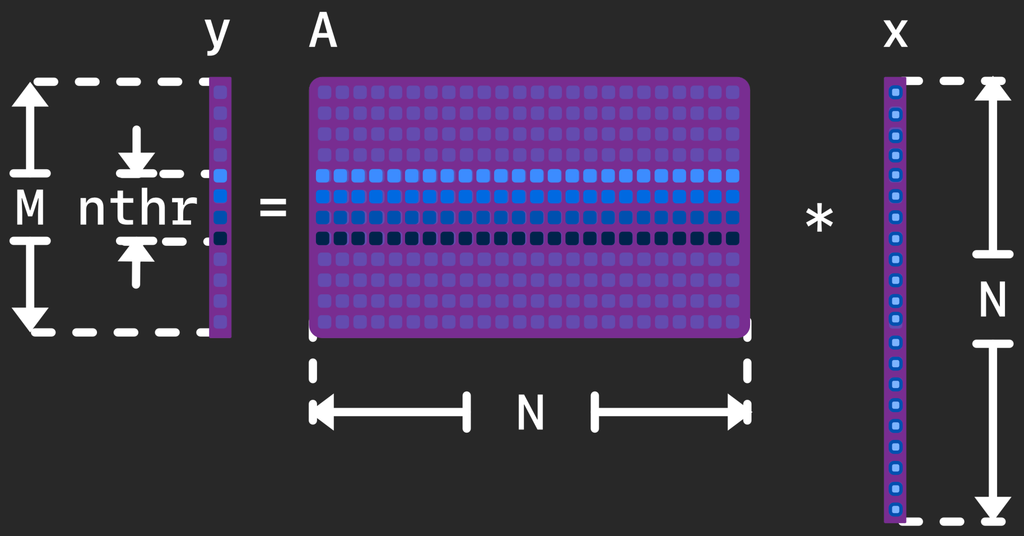
Matrix-vector multiplication

Our task is to modify the second call of the `Ax()` function to run on the GPU.



Matrix-vector multiplication

Our task is to modify the second call of the `Ax()` function to run on the GPU.

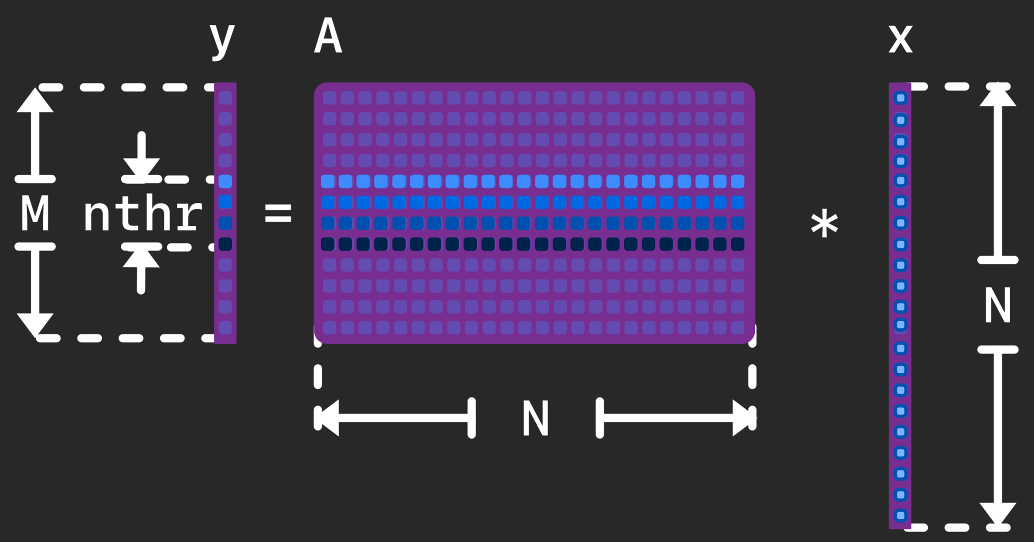


Straight-forward approach to begin with:

- Each block is responsible for one element of `y[]`
  - Each thread must read all elements of the corresponding row of `A[]`
  - Each thread must read all elements of `x[]`

Matrix-vector multiplication

Our task is to modify the second call of the `Ax()` function to run on the GPU.



Straight-forward approach to begin with:

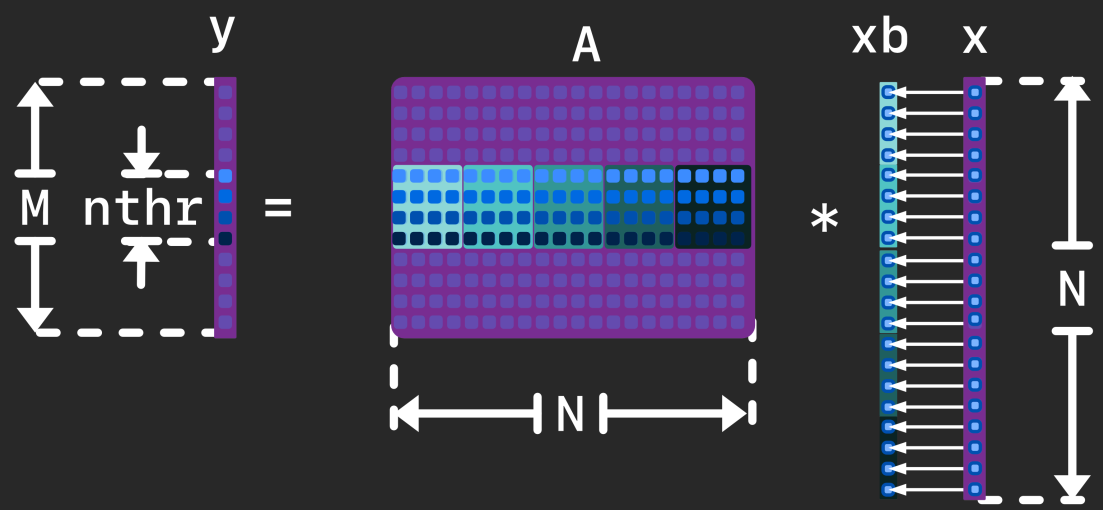
- Each block is responsible for one element of `y[]`
  - Each thread must read all elements of the corresponding row of `A[]`
  - Each thread must read all elements of `x[]`

E.g., using 256 GPU threads:

```
[ikoutsou@front02 ex02]$ srun -N 1  --cpus-per-task=16 -p p100  --gres=gpu:1 ./matvec 4096 8192
 CPU: nthr =   16   t0 = 0.0035 sec   P =  19.108 Gflop/s   B =  38.229 GB/s
 GPU: nthr =  256   t0 = 0.0020 sec   P =  32.994 Gflop/s   B =  66.013 GB/s
 Diff = 2.603650e-15
```

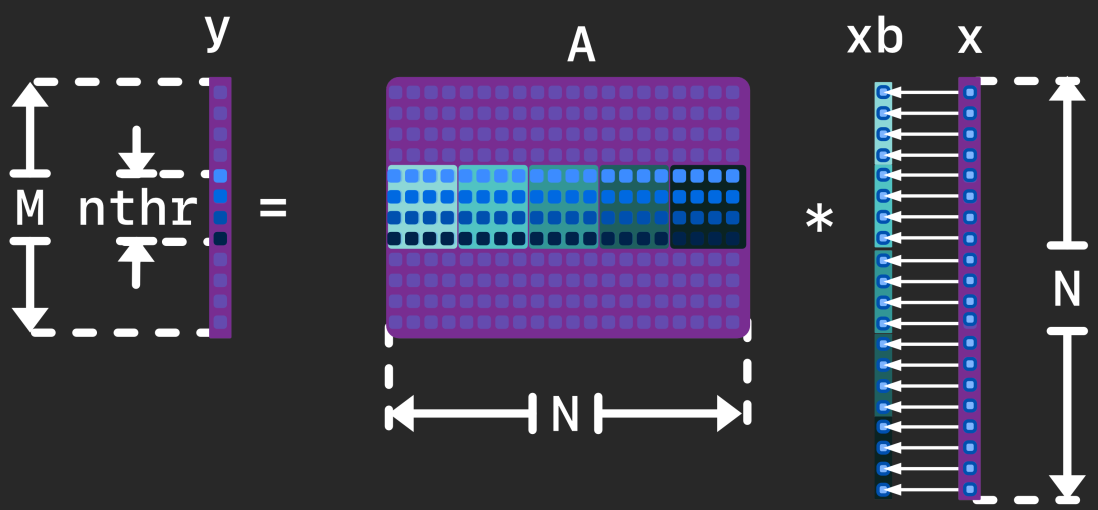
Matrix-vector multiplication

Now use a *shared array* to share the elements of `x[]`. Name the shared array `xb[]`:



Matrix-vector multiplication

Now use a *shared array* to share the elements of `x[]`. Name the shared array `xb[]`:

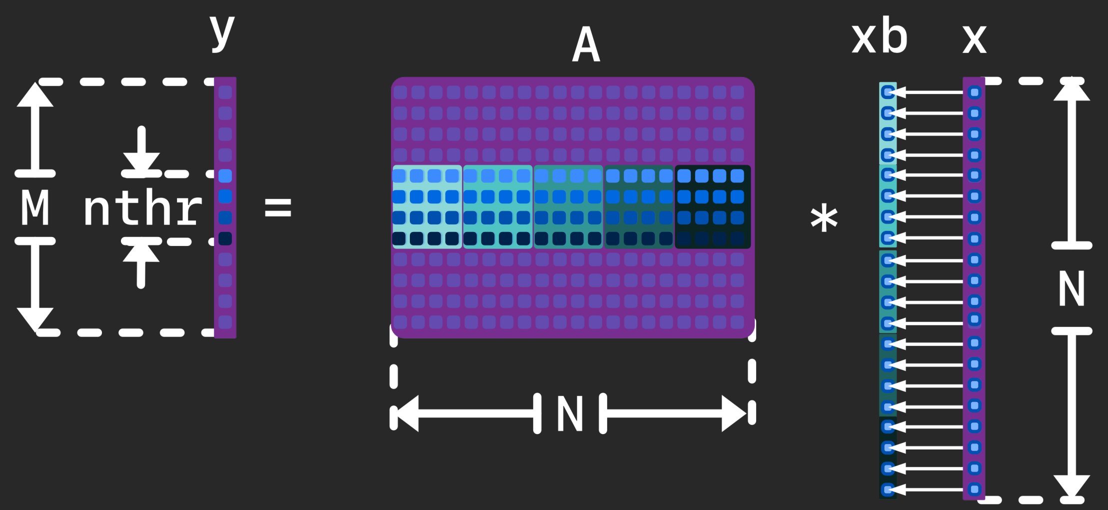


Notice that the shared array is of the size of the number of threads (`blockDim.x`) and therefore smaller than `x[]`

- Within each block, use all threads to read in the elements of `xb[]`
- This requires splitting the matrix-vector multiplication of the block into steps

Matrix-vector multiplication

Now use a *shared array* to share the elements of `x[]`. Name the shared array `xb[]`:



Notice that the shared array is of the size of the number of threads (`blockDim.x`) and therefore smaller than `x[]`

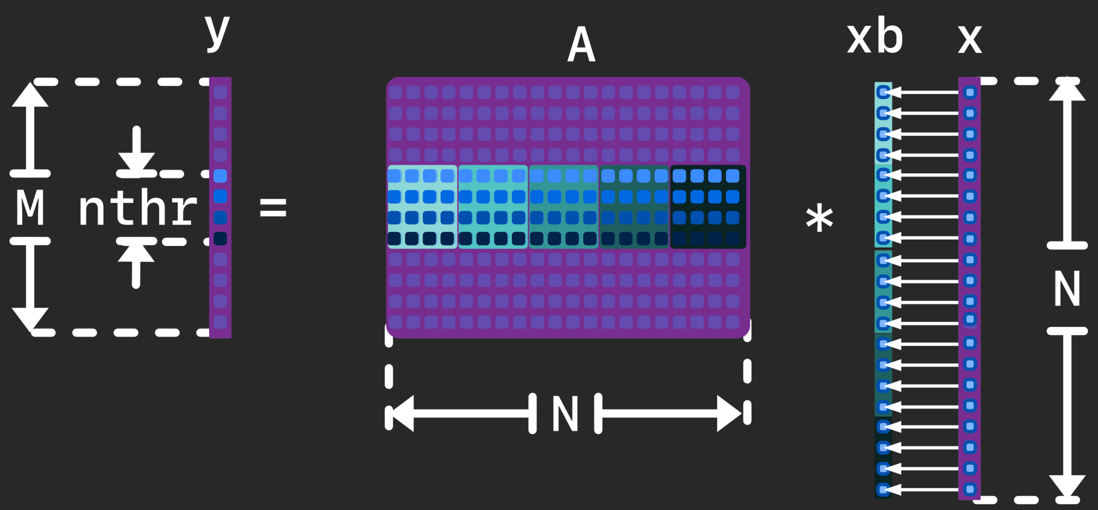
- Within each block, use all threads to read in the elements of `xb[]`
- This requires splitting the matrix-vector multiplication of the block into steps

Using 256 GPU threads:

```
[ikoutsou@front02 ex02]$ srun -N 1  --cpus-per-task=16 -p p100  --gres=gpu:1 ./matvec 4096 8192
 CPU: nthr =   16   t0 = 0.0035 sec   P =  19.384 Gflop/s   B =  38.782 GB/s
 GPU: nthr =  256   t0 = 0.0019 sec   P =  35.375 Gflop/s   B =  70.775 GB/s
 Diff = 2.603650e-15
```

Matrix-vector multiplication

Now use a *shared array* to share the elements of `x[]`. Name the shared array `xb[]`:



Notice that the shared array is of the size of the number of threads (`blockDim.x`) and therefore smaller than `x[]`

- Within each block, use all threads to read in the elements of `xb[]`
- This requires splitting the matrix-vector multiplication of the block into steps

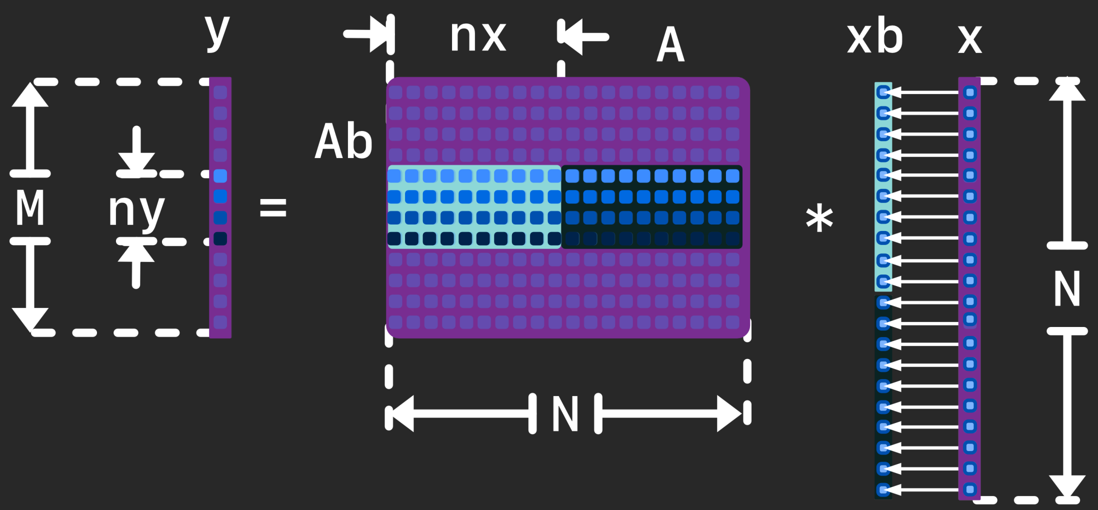
Using 256 GPU threads:

```
[ikoutsou@front02 ex02]$ srun -N 1  --cpus-per-task=16 -p p100  --gres=gpu:1 ./matvec 4096 8192
 CPU: nthr =   16   t0 = 0.0035 sec   P =  19.384 Gflop/s   B =  38.782 GB/s
 GPU: nthr =  256   t0 = 0.0019 sec   P =  35.375 Gflop/s   B =  70.775 GB/s
 Diff = 2.603650e-15
```

Not much improvement compared to previous version

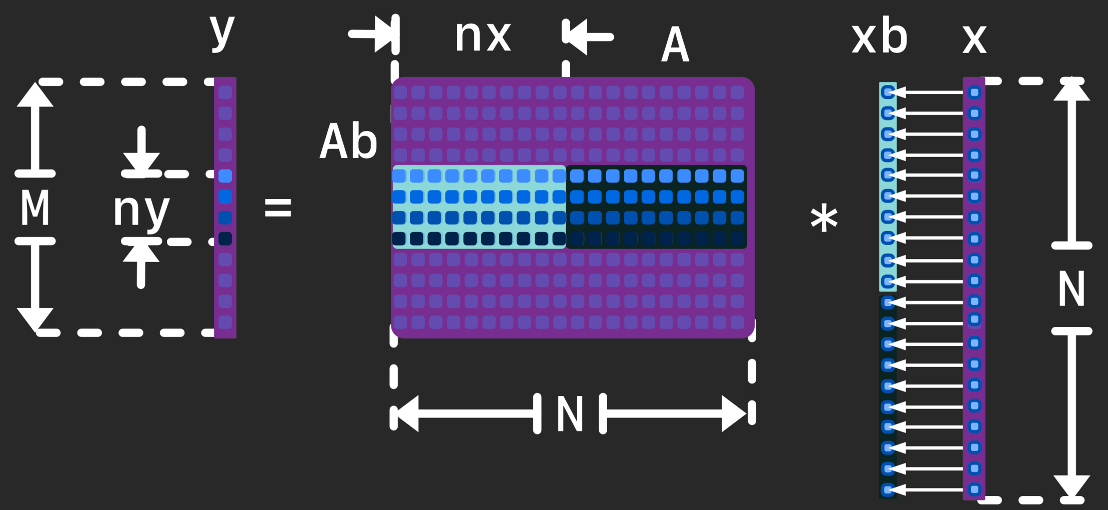
Matrix-vector multiplication

Now use a *shared array* for both `A[]` and `x[]`. Name them `Ab[]` and `xb[]`:



Matrix-vector multiplication

Now use a *shared array* for both `A[]` and `x[]`. Name them `Ab[]` and `xb[]`:

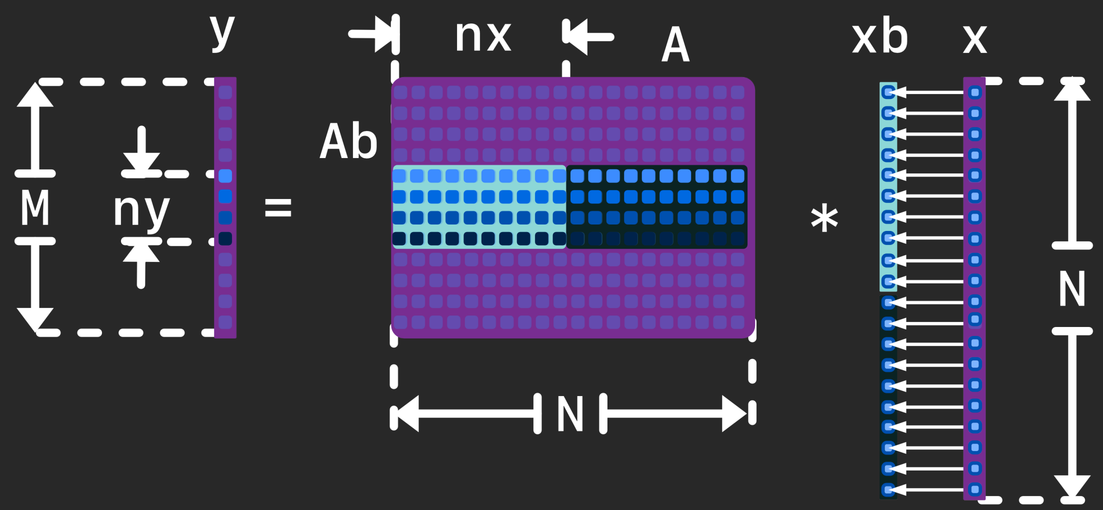


Use a 2-dimensional thread block

- All threads are used to fill in `Ab[]`
- Only some threads fill in `xb[]`
- Only some threads carry out the computation for `y[]`

Matrix-vector multiplication

Now use a *shared array* for both `A[]` and `x[]`. Name them `Ab[]` and `xb[]`:



Use a 2-dimensional thread block

- All threads are used to fill in `Ab[]`
- Only some threads fill in `xb[]`
- Only some threads carry out the computation for `y[]`

Using thread-blocks of, e.g. 8×64:

```
[ikoutsou@front02 ex02]$ srun -N 1  --cpus-per-task=8 -p nehalem  --gres=gpu:1 ./matvec 8 4 4096 8192
 CPU: nthr =    16         t0 = 0.0035 sec    P =  19.278 Gflop/s    B =  38.570 GB/s
 GPU: nthr = (   8,    4)  t0 = 0.0030 sec    P =  22.430 Gflop/s    B =  44.877 GB/s
 Diff = 2.603650e-15
```

Matrix-vector multiplication

Scanning for the optimal parameters:

```
GPU: nthr = (  4,   4)  t0 = 0.0054 sec   P =  12.460 Gflop/s   B =  24.928 GB/s
GPU: nthr = (  4,   8)  t0 = 0.0037 sec   P =  18.281 Gflop/s   B =  36.576 GB/s
GPU: nthr = (  4,  16)  t0 = 0.0019 sec   P =  35.321 Gflop/s   B =  70.668 GB/s
GPU: nthr = (  4,  32)  t0 = 0.0018 sec   P =  37.410 Gflop/s   B =  74.848 GB/s
GPU: nthr = (  4,  64)  t0 = 0.0020 sec   P =  33.173 Gflop/s   B =  66.371 GB/s
GPU: nthr = (  4, 128)  t0 = 0.0020 sec   P =  32.863 Gflop/s   B =  65.751 GB/s
GPU: nthr = (  4, 256)  t0 = 0.0035 sec   P =  19.077 Gflop/s   B =  38.167 GB/s
GPU: nthr = (  8,   4)  t0 = 0.0030 sec   P =  22.407 Gflop/s   B =  44.830 GB/s
GPU: nthr = (  8,   8)  t0 = 0.0020 sec   P =  33.205 Gflop/s   B =  66.433 GB/s
GPU: nthr = (  8,  16)  t0 = 0.0011 sec   P =  61.686 Gflop/s   B = 123.418 GB/s
GPU: nthr = (  8,  32)  t0 = 0.0013 sec   P =  53.139 Gflop/s   B = 106.316 GB/s
GPU: nthr = (  8,  64)  t0 = 0.0018 sec   P =  38.027 Gflop/s   B =  76.082 GB/s
GPU: nthr = (  8, 128)  t0 = 0.0020 sec   P =  33.138 Gflop/s   B =  66.300 GB/s
GPU: nthr = ( 16,   4)  t0 = 0.0015 sec   P =  43.351 Gflop/s   B =  86.733 GB/s
GPU: nthr = ( 16,   8)  t0 = 0.0013 sec   P =  53.179 Gflop/s   B = 106.396 GB/s
GPU: nthr = ( 16,  16)  t0 = 0.0010 sec   P =  67.793 Gflop/s   B = 135.635 GB/s
GPU: nthr = ( 16,  32)  t0 = 0.0013 sec   P =  52.475 Gflop/s   B = 104.988 GB/s
GPU: nthr = ( 16,  64)  t0 = 0.0017 sec   P =  39.566 Gflop/s   B =  79.162 GB/s
GPU: nthr = ( 32,   4)  t0 = 0.0014 sec   P =  47.530 Gflop/s   B =  95.096 GB/s
GPU: nthr = ( 32,   8)  t0 = 0.0015 sec   P =  44.707 Gflop/s   B =  89.447 GB/s
GPU: nthr = ( 32,  16)  t0 = 0.0016 sec   P =  42.150 Gflop/s   B =  84.330 GB/s
GPU: nthr = ( 32,  32)  t0 = 0.0019 sec   P =  35.967 Gflop/s   B =  71.960 GB/s
GPU: nthr = ( 64,   4)  t0 = 0.0013 sec   P =  52.426 Gflop/s   B = 104.890 GB/s
GPU: nthr = ( 64,   8)  t0 = 0.0015 sec   P =  45.561 Gflop/s   B =  91.155 GB/s
GPU: nthr = ( 64,  16)  t0 = 0.0017 sec   P =  38.771 Gflop/s   B =  77.570 GB/s
GPU: nthr = (128,   4)  t0 = 0.0013 sec   P =  52.260 Gflop/s   B = 104.559 GB/s
GPU: nthr = (128,   8)  t0 = 0.0016 sec   P =  42.960 Gflop/s   B =  85.952 GB/s
GPU: nthr = (256,   4)  t0 = 0.0015 sec   P =  45.502 Gflop/s   B =  91.037 GB/s
```

~130 GB/s is about the maximum we can obtain

Matrix-vector multiplication

Now let's see what we get when using CUDA's implementation of the same kernel

- The matrix-vector multiplication is implemented as part of CUDA's BLAS implementation

```
#include <cublas_v2.h>
```

- The function to use is `cublasSgemv()` — see: https://docs.nvidia.com/cuda/cublas/index.html#cublas-lt-t-gt-gemv

Matrix-vector multiplication

Now let's see what we get when using CUDA's implementation of the same kernel

- The matrix-vector multiplication is implemented as part of CUDA's BLAS implementation

```
#include <cublas_v2.h>
```

- The function to use is `cublasSgemv()` — see: https://docs.nvidia.com/cuda/cublas/index.html#cublas-lt-t-gt-gemv
- This function is general and computes: $y = \alpha A x + \beta y$, where α and β are scalars

Matrix-vector multiplication

Now let's see what we get when using CUDA's implementation of the same kernel

- The matrix-vector multiplication is implemented as part of CUDA's BLAS implementation

```
#include <cublas_v2.h>
```

- The function to use is `cublasSgemv()` — see: https://docs.nvidia.com/cuda/cublas/index.html#cublas-lt-t-gt-gemv
- This function is general and computes: $y = \alpha A x + \beta y$, where α and β are scalars
- In our case, we need: $\alpha = 1$ and $\beta = 0$.

Matrix-vector multiplication

Call the CUBLAS function via:

```
cublasSgemv(handle, CUBLAS_OP_T, n, m, &alpha, d_A, n, d_x, 1, &beta, d_y, 1);
```

Matrix-vector multiplication

Call the CUBLAS function via:

```
cublasSgemv(handle, CUBLAS_OP_T, n, m, &alpha, d_A, n, d_x, 1, &beta, d_y, 1);
```

- `CUBLAS_OP_T` means transpose `A`, because CUBLAS expects matrices with the row index running fastest

Matrix-vector multiplication

Call the CUBLAS function via:

```
cublasSgemv(handle, CUBLAS_OP_T, n, m, &alpha, d_A, n, d_x, 1, &beta, d_y, 1);
```

- `CUBLAS_OP_T` means transpose `A`, because CUBLAS expects matrices with the row index running fastest
- `handle` is just the CUBLAS context:

```
cublasHandle_t handle;
cublasCreate(&handle);
```

- Add `-lcublas` to the compile command

Matrix-vector multiplication

Call the CUBLAS function via:

```
cublasSgemv(handle, CUBLAS_OP_T, n, m, &alpha, d_A, n, d_x, 1, &beta, d_y, 1);
```

- `CUBLAS_OP_T` means transpose `A`, because CUBLAS expects matrices with the row index running fastest
- `handle` is just the CUBLAS context:

```
cublasHandle_t handle;
cublasCreate(&handle);
```

- Add `-lcublas` to the compile command

Now CUBLAS chooses the number of threads:

```
[ikoutsou@front02 ex02]$ srun -N 1  --cpus-per-task=8 -p nehalem  --gres=gpu:1 ./matvec 4096 8192
 CPU: nthr =   16        t0 = 0.0037 sec   P =  18.241 Gflop/s   B =  36.495 GB/s
 GPU:                    t0 = 0.0037 sec   P =  17.944 Gflop/s   B =  35.902 GB/s
 Diff = 1.380096e-12
```

Matrix-vector multiplication

Call the CUBLAS function via:

```
cublasSgemv(handle, CUBLAS_OP_T, n, m, &alpha, d_A, n, d_x, 1, &beta, d_y, 1);
```

- `CUBLAS_OP_T` means transpose `A`, because CUBLAS expects matrices with the row index running fastest
- `handle` is just the CUBLAS context:

```
    cublasHandle_t handle;
    cublasCreate(&handle);
```

- Add `-lcublas` to the compile command

Now CUBLAS chooses the number of threads:

```
[ikoutsou@front02 ex02]$ srun -N 1  --cpus-per-task=8 -p nehalem  --gres=gpu:1 ./matvec 4096 8192
 CPU: nthr =   16        t0 = 0.0037 sec   P =  18.241 Gflop/s   B =  36.495 GB/s
 GPU:                    t0 = 0.0037 sec   P =  17.944 Gflop/s   B =  35.902 GB/s
 Diff = 1.380096e-12
```

NVIDIA's version is not necessarily faster than our hand-tuned version