Introduction to High Performance Computing

\\

SDS406 – Fall semester, 2024 - 2025

\\

L07: GPU programming, 11th November 2024

Outline

Matrix-vector multiplication on GPUs

Covering:

- Shared memory
- Details of GPU thread scheduling and warps
- Use of `__syncthreads()`
- Two-dimensional thread blocks

Matrix-vector multiplication

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

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

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

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/l07/ex01/.` for the CPU code[1]:

```
[ikoutsou@front02 l07]$ cp -r /onyx/data/sds406f24/l07/ex01 .
[ikoutsou@front02 l07]$ cd ex01/.
[ikoutsou@front02 ex01]$ nvcc -arch=sm_60 -O3 -Xcompiler -fopenmp -o matvec matvec.cu
[ikoutsou@front02 ex01]$ export OMP_PLACES="cores"
[ikoutsou@front02 ex01]$ export OMP_PROC_BIND="close"
[ikoutsou@front02 ex01]$ export OMP_NUM_THREADS=16
[ikoutsou@front02 ex01]$ 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
```

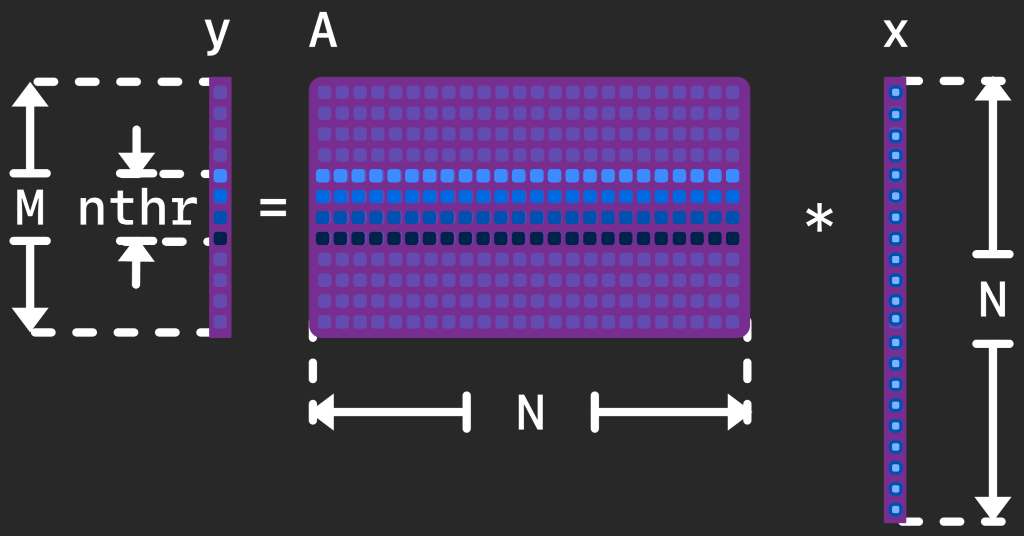
[1] Same as last week's: `/onyx/data/sds406f24/l06/ex02/`

Matrix-vector multiplication

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

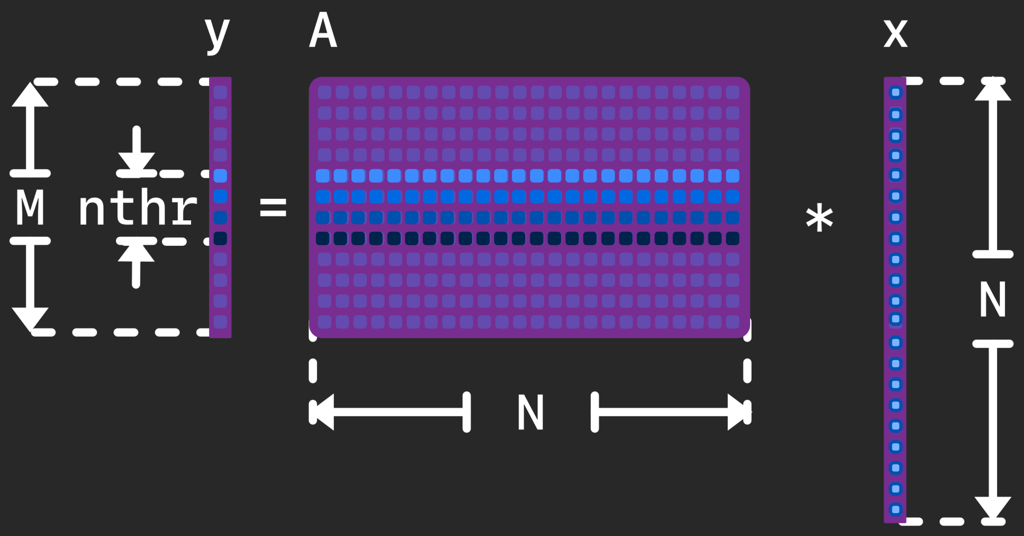
Matrix-vector multiplication

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.

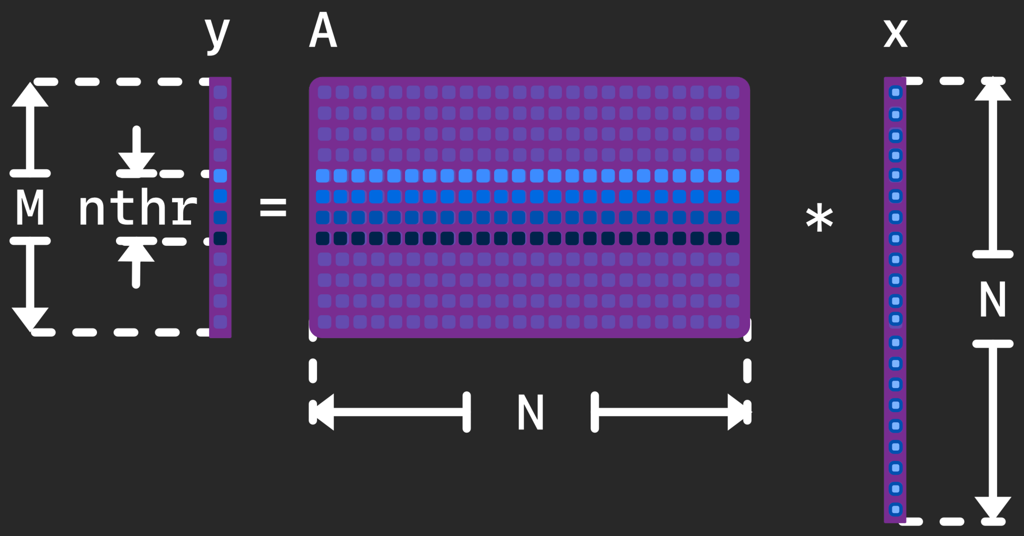


Straight-forward approach to begin with:

- 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:

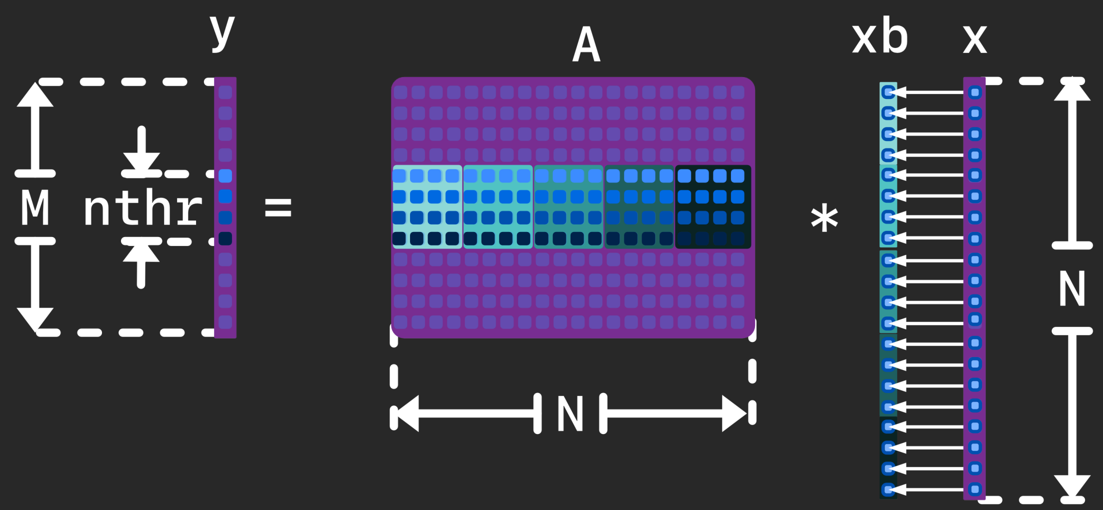
- Each block is responsible for one element of `y[]`
  - Each thread must read all elements of the corresponding row of `A[]`
  - Each thread must read all elements of `x[]`

E.g., using 256 GPU threads:

```
[ikoutsou@front02 ex01]$ 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
```

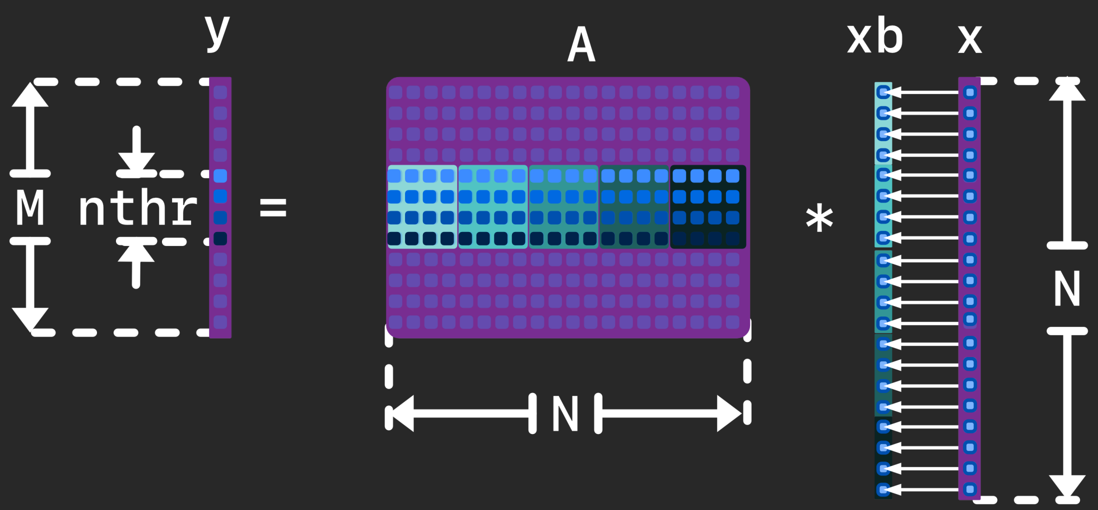
Matrix-vector multiplication

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[]`:

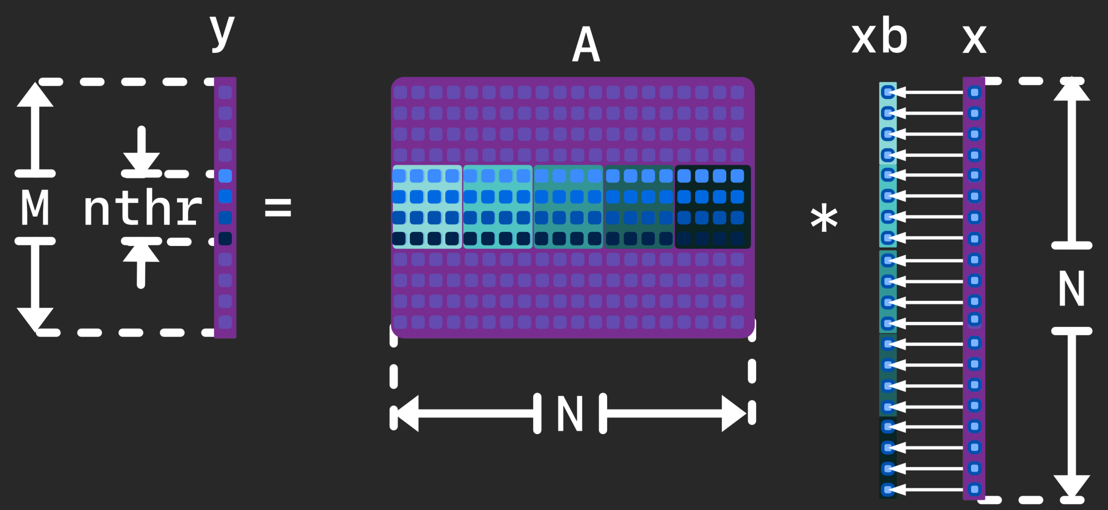


Notice that the shared array is of the size of the number of threads (`blockDim.x`) and therefore smaller than `x[]`

- 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[]`

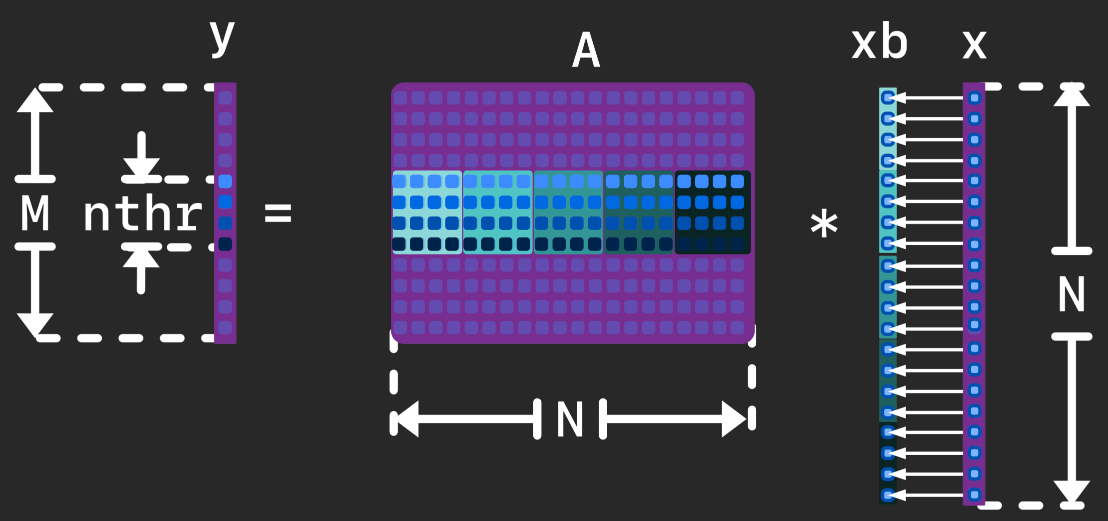
- Within each block, use all threads to read in the elements of `xb[]`
- This requires splitting the matrix-vector multiplication of the block into steps

Using 256 GPU threads:

```
[ikoutsou@front02 ex01]$ 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

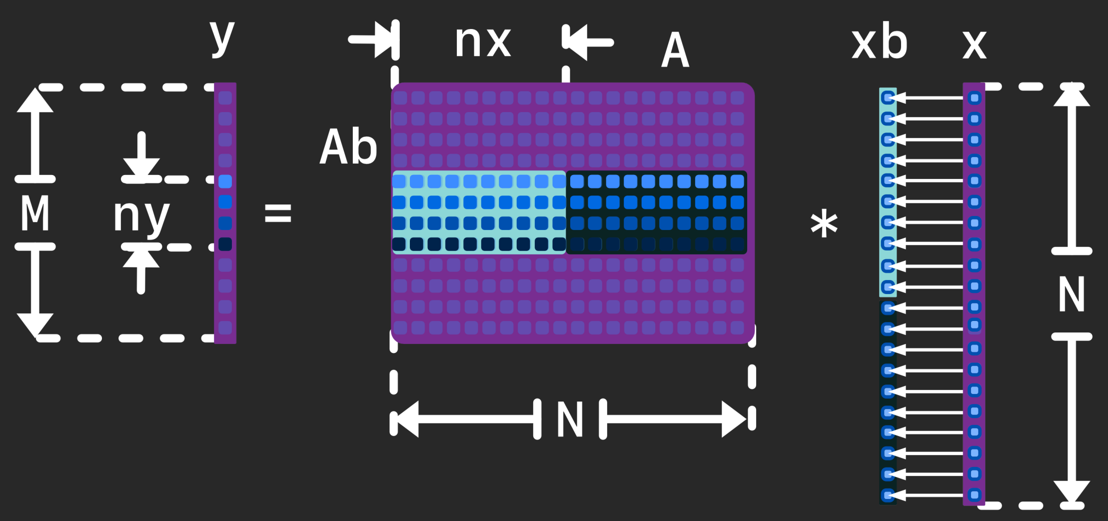
Using 256 GPU threads:

```
[ikoutsou@front02 ex01]$ 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

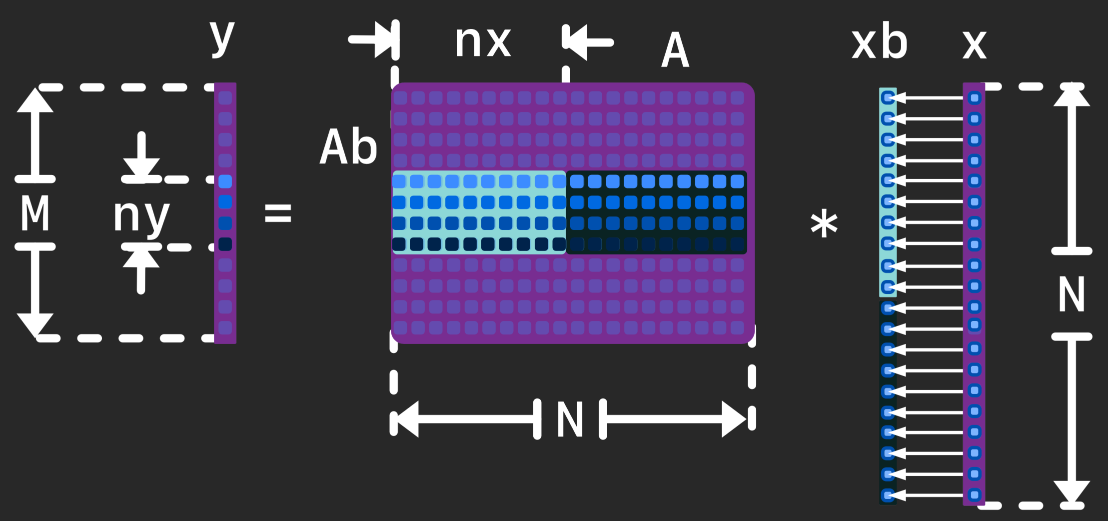
Matrix-vector multiplication

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[]`:

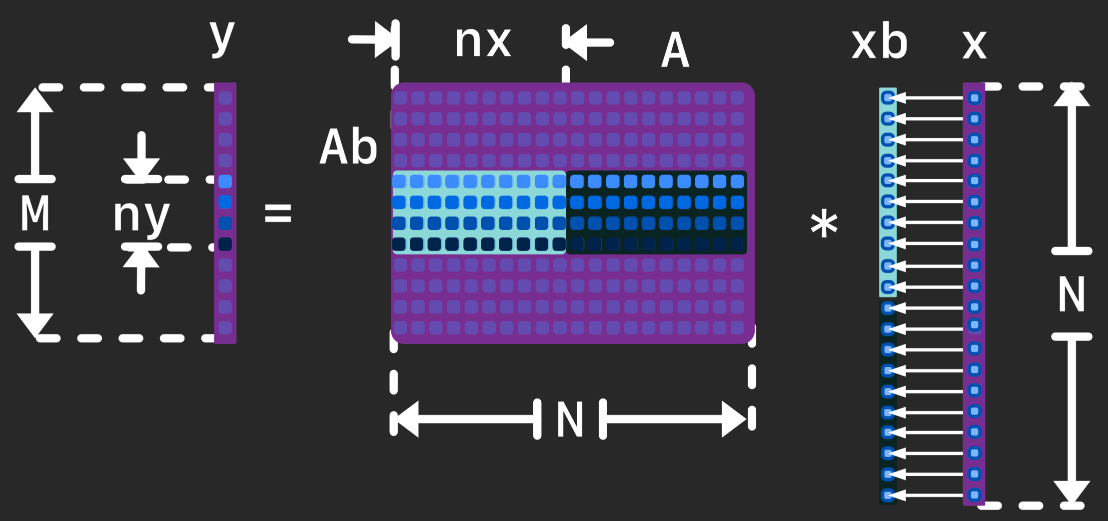


Use a 2-dimensional thread block

- 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 ex01]$ 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 ex01]$ 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 ex01]$ 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