SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

GPU Programming

Sivasankar Arul, IT4Innovations

June/2021

Univerza v Ljubljani

TECHNISCHE UNIVERSITÄT WIEN

CINECA consorzio interuniversitario

VSB TECHNICAL UNIVERSITY OF OSTRAVA | IT4INNOVATIONS NATIONAL SUPERCOMPUTING CENTER

Co-funded by the Erasmus+ Programme of the European Union

This project has been funded with support from the European Commission.
This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Matrix Vector Product

Matrix – Vector Multiplication

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \qquad x_n = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

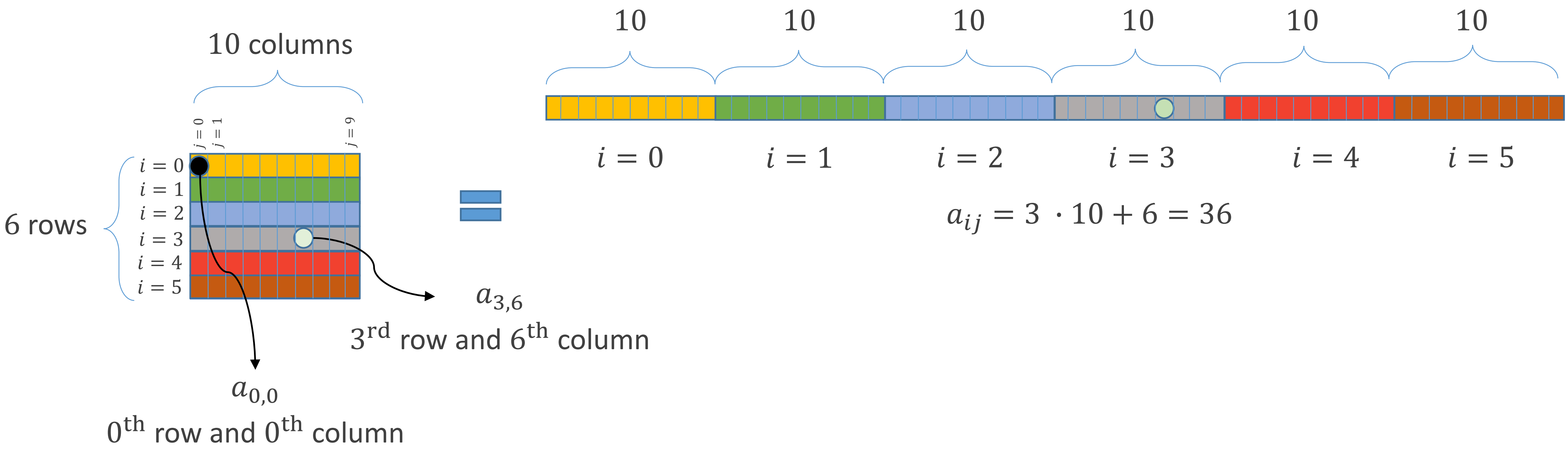
Sequential algorithm for Matrix - Vector product

```
for i = 1, 2, ..., m do
    out[i] = 0
    for j = 1, 2, ..., n do
        out[i]+ = mat[i][j]* x[j]
    end for
end for
```

$$Ax = \begin{pmatrix} a_{1,1}x_1 + a_{1,2}x_2 + \cdots & a_{1,n}x_n \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots & a_{2,n}x_n \\ \vdots & \\ a_{m,1}x_1 + a_{m,2}x_2 + \cdots & a_{m,n}x_n \end{pmatrix}$$

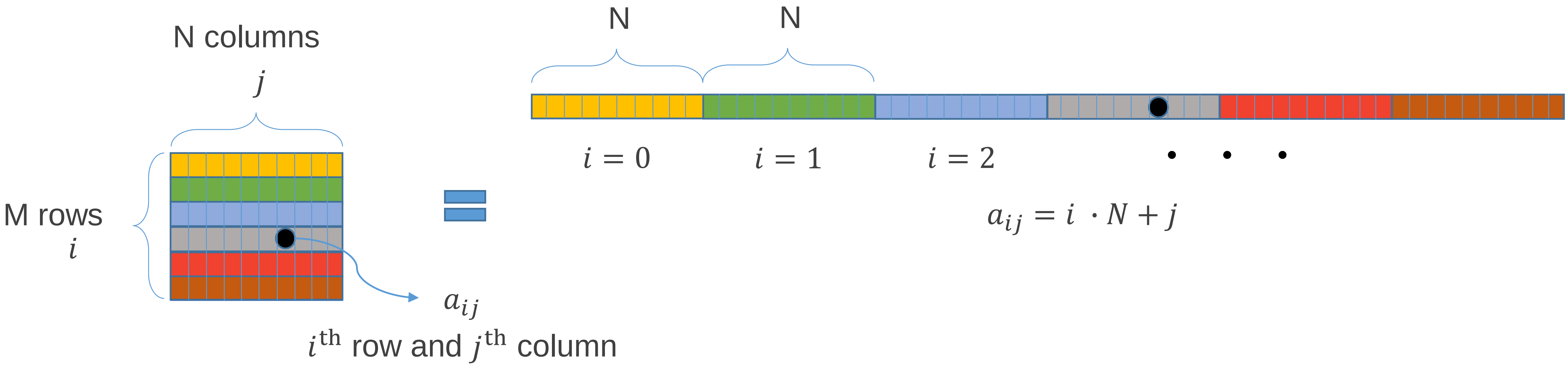
Matrix Vector Product

The matrix is stored as an array.



Matrix Vector Product

The matrix is stored as an array.



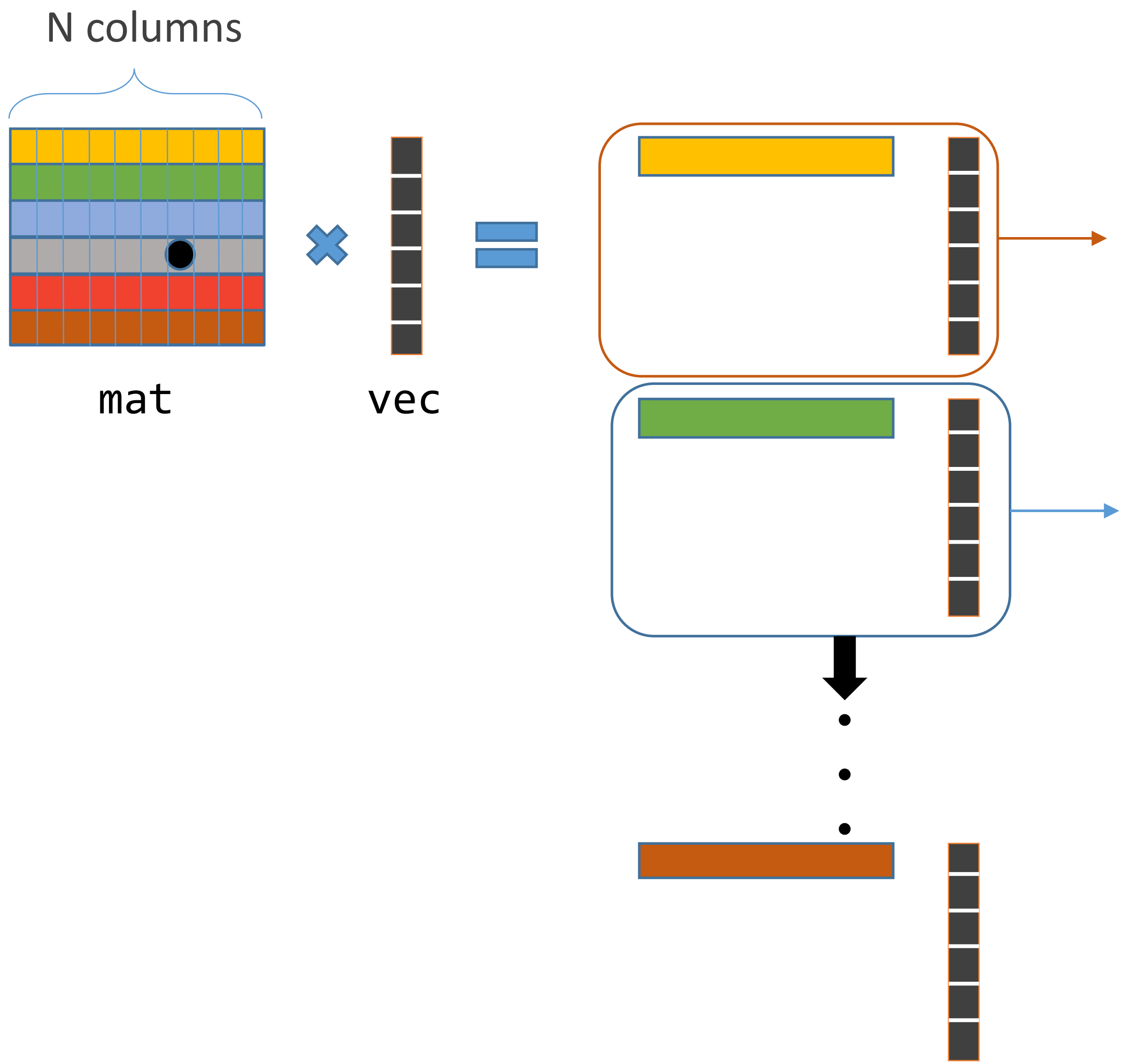
$$a_{ij} = i \cdot N + j$$

To access a particular row :

```
for(int col=0; col<N; col++)
    mat[row*N+col];
```

Since `col = 0, 1, 2, ..., N-1`

Matrix Vector Product



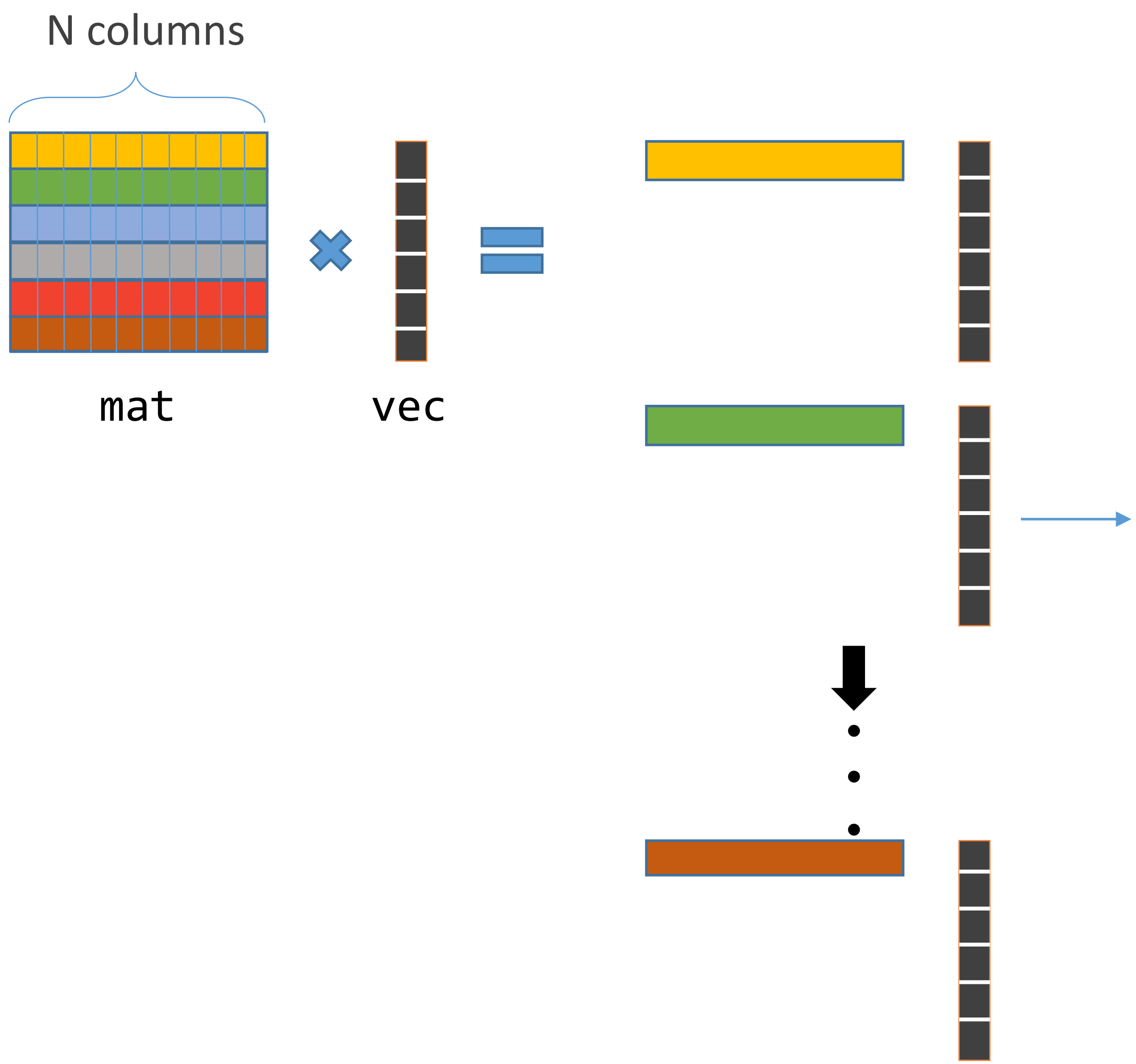

```
row = 0
for(int col=0; col<N; col++){
      sum=sum + mat[row*N+col]* vec[col];}
```

```
row = 1
for(int col=0; col<N; col++){
      sum=sum + mat[row*N+col]* vec[col];}
```

Here col = 0, 1, 2, ..., N-1

We take the row of the matrix, and find its dot product with the vector.

Matrix Vector Product




```
float sum;
for(int row = 0; row < N; row++){
   sum = 0;

   for(int col = 0; col < N; col++){
      sum=sum + mat[row*N + col]*vec[col];}

   out[row] = sum;}

}
```

To find matrix vector product:

- We take each row of the matrix
- Then we find its dot product with the vector

Matrix Vector Product - C

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```c
#include <stdio.h>
#include <iostream>
#include <cuda.h>
#include <time.h>
#include <sys/time.h>
```

The “include” statements

```c
int main(void){


}
```

The main body of the code.

```c
double t;
int N = 32768;
int M = N;
```

“N” and “M” are the size of the matrix
“t” is the variable to store the time

Matrix Vector Product - C

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
float *a, *b, *c, *d;
float *dev_a, *dev_b, *dev_c;

double t;
int N = 32768;
int M = N;
a = (float *)malloc(sizeof(float)*N);
b = (float *)malloc(sizeof(float)*N*M);
c = (float *)malloc(sizeof(float)*M);
```

➡

```
init_array(a, N);
init_mat(b, N, M);
init_array(c, M);
```

```
void init_array(float *a, const int N) {
        int i;
        for(i=0; i<N; i++)
                a[i] = 1.0;
}

void init_mat(float *a, const int N, const int M) {
        int i, j;
        for(i=0; i<N; i++)
            for(j=0; j<M; j++)
                    a[i*M+j] = 2.0;
}
```

Memory Allocation for the variables

Initializing the variables

Matrix Vector Product - C

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
for (row = 0; row < N; row++){
    sum = 0.0;
    for (col = 0; col < N; col++){
        sum+= b[row*N  + col]*a[col];
    }
    c[row] =sum;
}
```

Matrix Vector product

➡

```
free(a);  free(b); free(c);
```

Deallocation of Memory

EXERCISE 2 : Matrix Vector product using GPU program

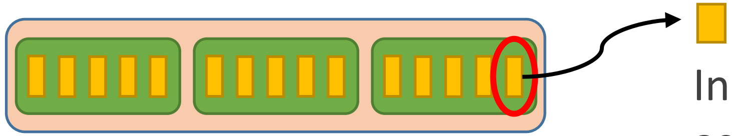
Source code

FOLDER: EX2_MATRIXVECTMUL

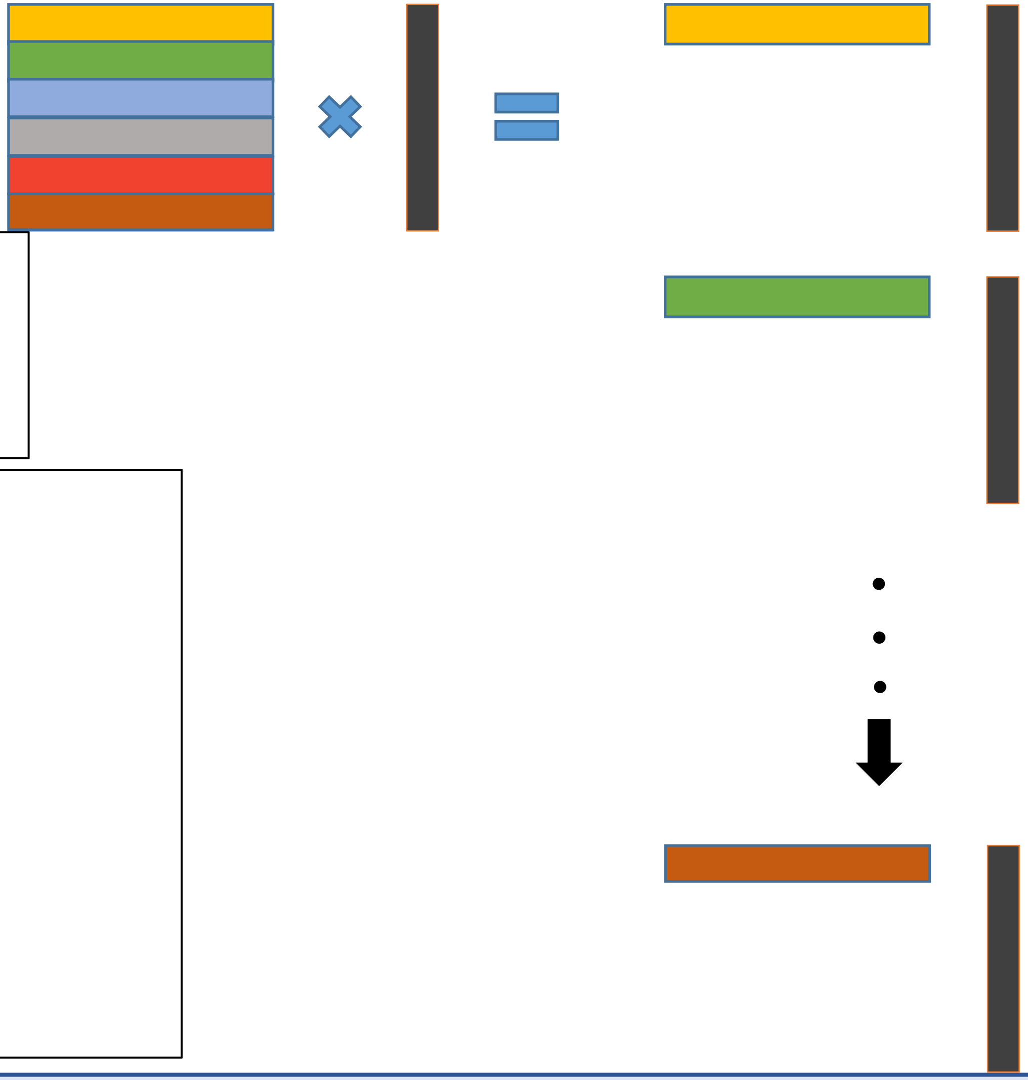
Product – One Core

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

One core



In one core
as one thread

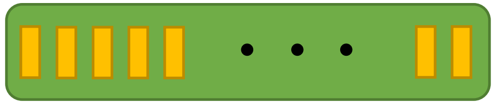


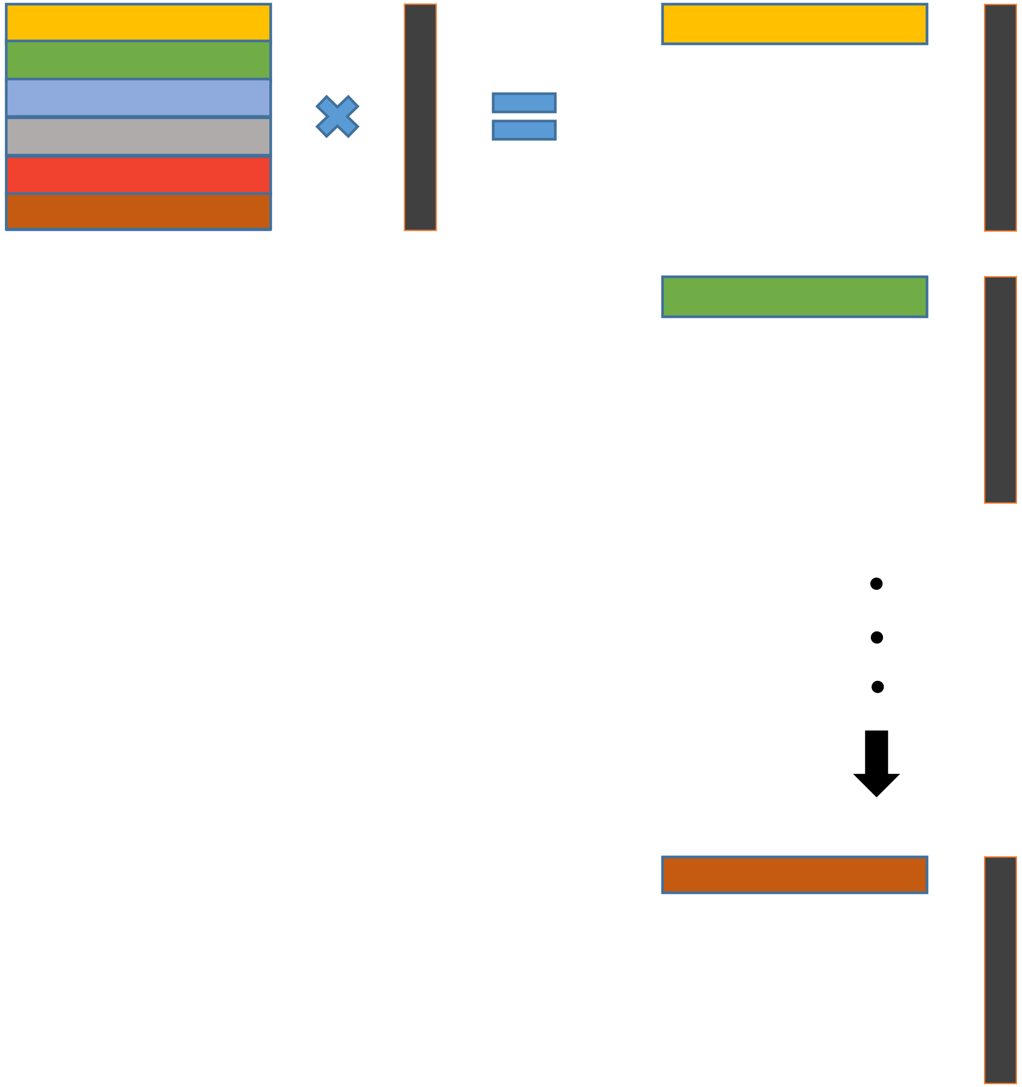
```
// Main function
int block_size = 1;
int grid_size  = 1;
matvec<<<grid_size, block_size>>>(dev_a, dev_b, dev_c, N, M);
```

```
__global__ void matvec(float *vec, float *mat,
        float *out, const int N, const int M)
{
    float sum;
    for(int row = 0; row < N; row++){
       sum = 0;

       for(int col = 0; col < N; col++){
          sum=sum + mat[row*N + col] * vec[col];}

       out[row] = sum;}  }
```

Product – One SM

One streaming multiprocessor



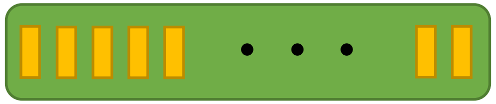


```
matvec <<<1,256>>> (dev_a, dev_b, dev_c, N, M);
```

Each thread performs the multiplication on a certain chunk of the rows of the matrix.

Product – One SM

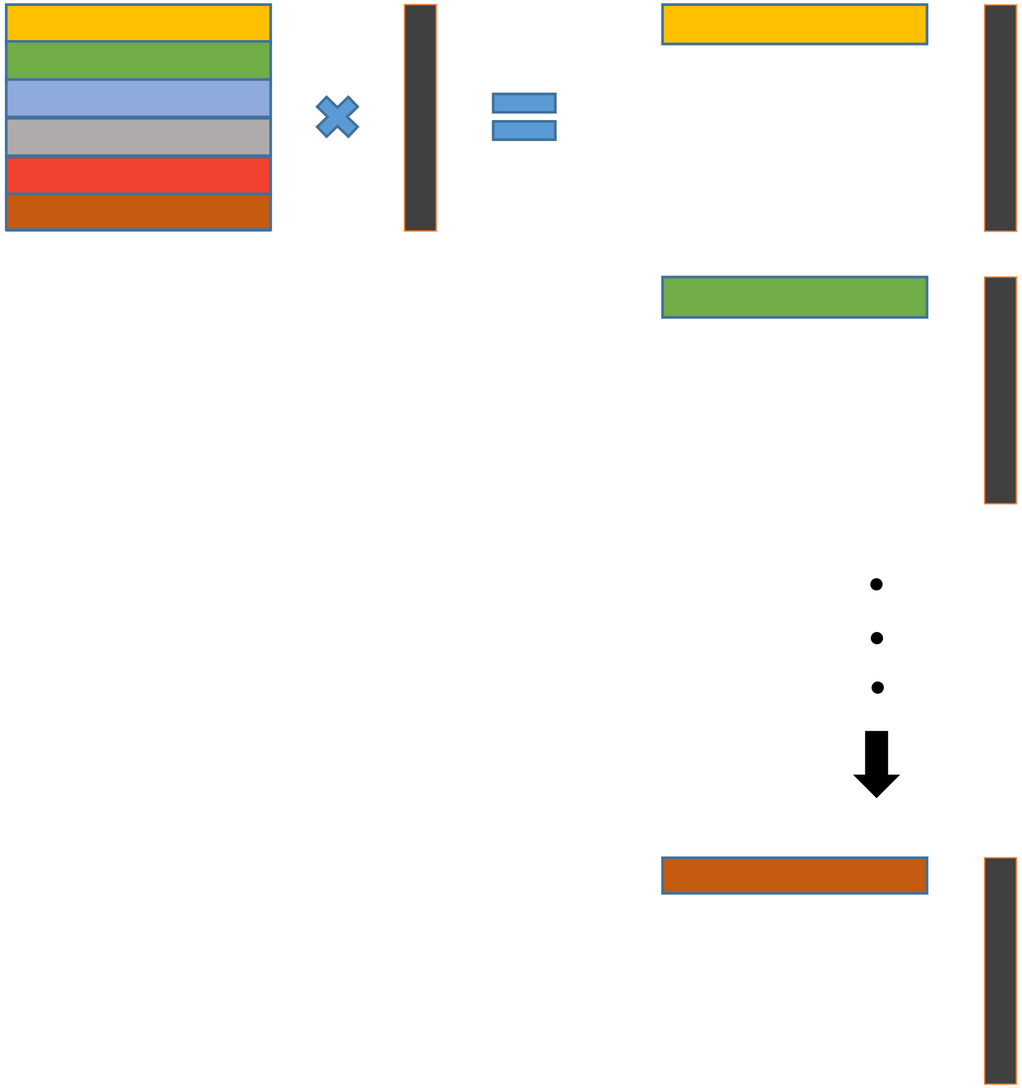
One streaming multiprocessor



```
int index  = threadIdx.x;
int stride = blockDim.x;

for(int row=index; row<N; row+=stride)
```

In vector addition, the index and stride is used to distribute the vector element among the threads.

Here, they are used to distribute the rows of the matrix among the threads.



Product – One SM

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
for(int row=index; row<N; row+=stride){
   sum = 0;
   for(int col = 0; col < N;col++)
       sum += vec[col] * mat[(row * N) + col];

   out[row] = sum;}
```

thread 0

Local variables for this thread:

```
index  = threadIdx.x = 0
stride = blockDim.x  = 256
```

For loop

first loop:

```
row = index = 0
out[row = 0]   = mat[row = 0] · vec
```

second loop:

```
row = row + stride = 256
out[row = 256] = mat[row = 256] · vec
```

third loop:

```
row = row + stride = 512
out[row = 512] = mat[row = 512] · vec
```

until: `row < n`

thread 1

Local variables for this thread:

```
index  = threadIdx.x = 1
stride = blockDim.x  = 256
```

For loop

first loop:

```
row = index = 1
out[row = 1]     = mat[row = 1] · vec
```

second loop:

```
row = row + stride = 257
out[row = 257]   = mat[row = 257] · vec
```

third loop:

```
row = row + stride = 513
out[row = 513]   = mat[row = 513] · vec
```

until: `row < n`

Product – One SM

1st loop

threadIdx.x	0	1	2	255
row	0	1	2	255

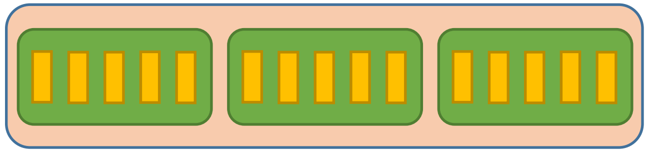
2nd loop

threadIdx.x	0	1	2	255
row	256	257	258	511

```
__global__ void matvec(float *vec, float *mat, float *out,
const int N, const int M){

 int index  = threadIdx.x;
 int stride = blockDim.x;
        float sum = 0;
        for(int row=index; row<N; row+=stride){
            sum = 0;
                for(int col = 0; col<N;col++){
                sum += vec[col]*mat[(row*N)+col];
                }
            out[row]=sum;
            }
}
```

Product – One SM

CUDA-enabled GPU



Many streaming multiprocessors can be used

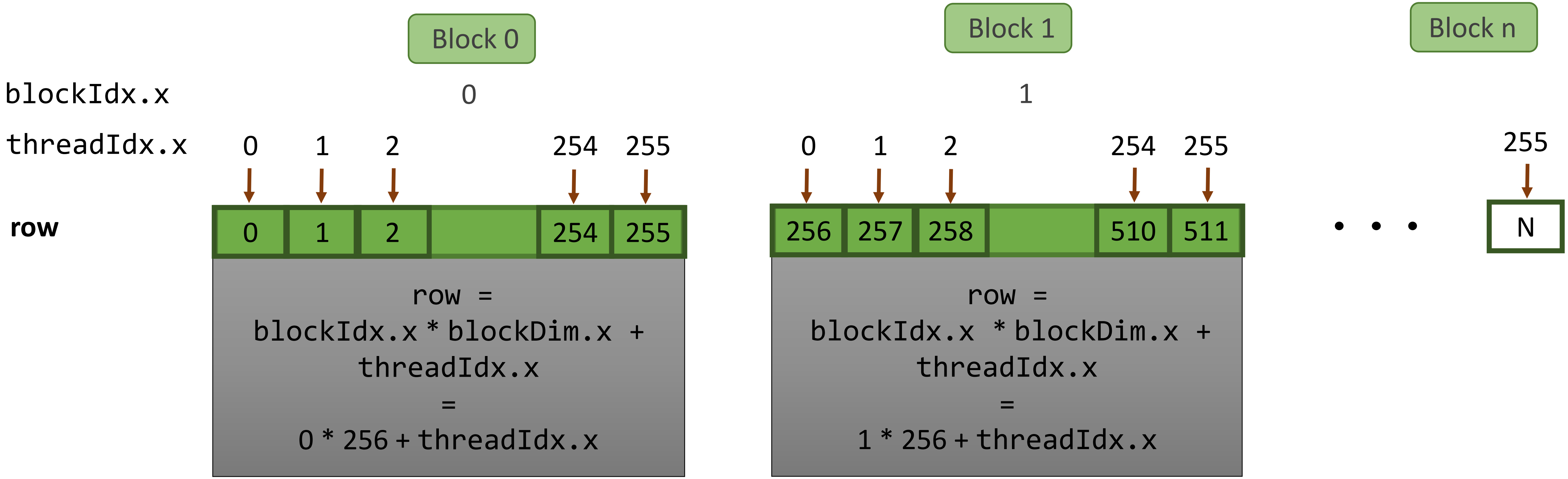
Each thread performs the multiplication of one row of the matrix with the vector

```
__global__ void matvec(float *vec, float *mat, float
*out, const int N, const int M){

    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    float sum = 0;

    for(int i = 0; i < N; i++)
        sum += vec[i] * mat[(tid * M) + i];

     out[tid] = sum;
}
```

Product - Multiple Blocks

Each thread can have its local variables. We define three local variables:

1. The id of the thread : `threadIdx.x`
2. The number of threads in the block : `blockDim.x = 256`
3. The block to which the thread belongs to : `blockIdx.x`



Programming Steps

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
#include <stdio.h>
#include <iostream>
#include <cuda.h>
#include <time.h>
#include <sys/time.h>
```

The “include” statements

```
int main(void){


}
```

The main body of the code.

```
double t;
int N = 32768;
int M = N;
```

“N” and “M” are the size of the matrix
“t” is the variable to store the time

Programming Steps

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
float *a, *b, *c, *d;

init_array(a, N)
init_mat(a, N)
init_array(a, N)
```

```
void init_array(float *a, const int N) {
        int i;
        for(i = 0; i < N; i++)
                a[i] = 1.0;
}

void init_mat(float *a, const int N, const int M) {
        int i, j;
        for(i = 0; i < N; i++)
            for(j = 0; j < M; j++)
                    a[I * M + j] = 2.0;
}
```

Memory Allocation of variable in device and initialization

Programming Steps

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
float *dev_a, *dev_b, *dev_c;

cudaMalloc((void **)&dev_a, sizeof(float)*N);
cudaMalloc((void **)&dev_b, sizeof(float)*N*M);
cudaMalloc((void **)&dev_c, sizeof(float)*M);
```

Memory Allocation in device

⬇

```
cudaMemcpy(dev_a, a, sizeof(float)*N,   cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, sizeof(float)*N*M, cudaMemcpyHostToDevice);
```

Copy variable from host to device

Programming Steps


```
cudaMemcpy(c, dev_c, sizeof(float)*M, cudaMemcpyDeviceToHost);
```

Data transfer from Device to Host

⬇

```
cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);

free(a);
free(b);
free(c);
free(d);
```

Deallocation of Memory

Computation – One thread

Computation in Device

```
__global__ void matvec(float *vec, float *mat,
        float *out, const int N, const int M)
{
    float sum;
    for(int row = 0; row < N; row++){
       sum = 0;

       for(int col = 0; col < N; col++){
          sum = sum + mat[row*N + col] * vec[col];}

       out[row] = sum;}
}
```

```
// Main function
int block_size = 1;
int grid_size  = 1;
matvec<<<grid_size, block_size>>>(dev_a, dev_b, dev_c, N, M);
cudaDeviceSynchronize();
```

Computation – One SM

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Computation in Device

```
__global__ void matvec(float *vec, float *mat, float *out,
const int N, const int M){

 int index  = threadIdx.x;
 int stride = blockDim.x;
        float sum=0;
        for(int row = index; row < N; row += stride){
            sum = 0;
                for(int col = 0; col < N;col++){
                sum += vec[col] * mat[(row*N) + col];
                }
            out[row] = sum;            }
}
```

```
// Main function
int block_size = 256;
int grid_size  = 1;
matvec<<<grid_size, block_size>>>(dev_a, dev_b, dev_c, N, M);
cudaDeviceSynchronize();
```

Computation – Multiple SMs

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Computation in Device

```
__global__ void matvec(float *vec, float *mat, float *out,
const int N, const int M){

    int row   = threadIdx.x + blockIdx.x * blockDim.x;
    float sum = 0;

    for(int col = 0; col < N; col++)
        sum += vec[col]*mat[(row*M) + col];

    out[tid] = sum;
}
```

```
// Main function
int blocksize = 256;
int nblocks   = N / blocksize;
matvec<<<grid_size,block_size>>>(dev_a, dev_b, dev_c, N, M);
cudaDeviceSynchronize();
```

Programming steps

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

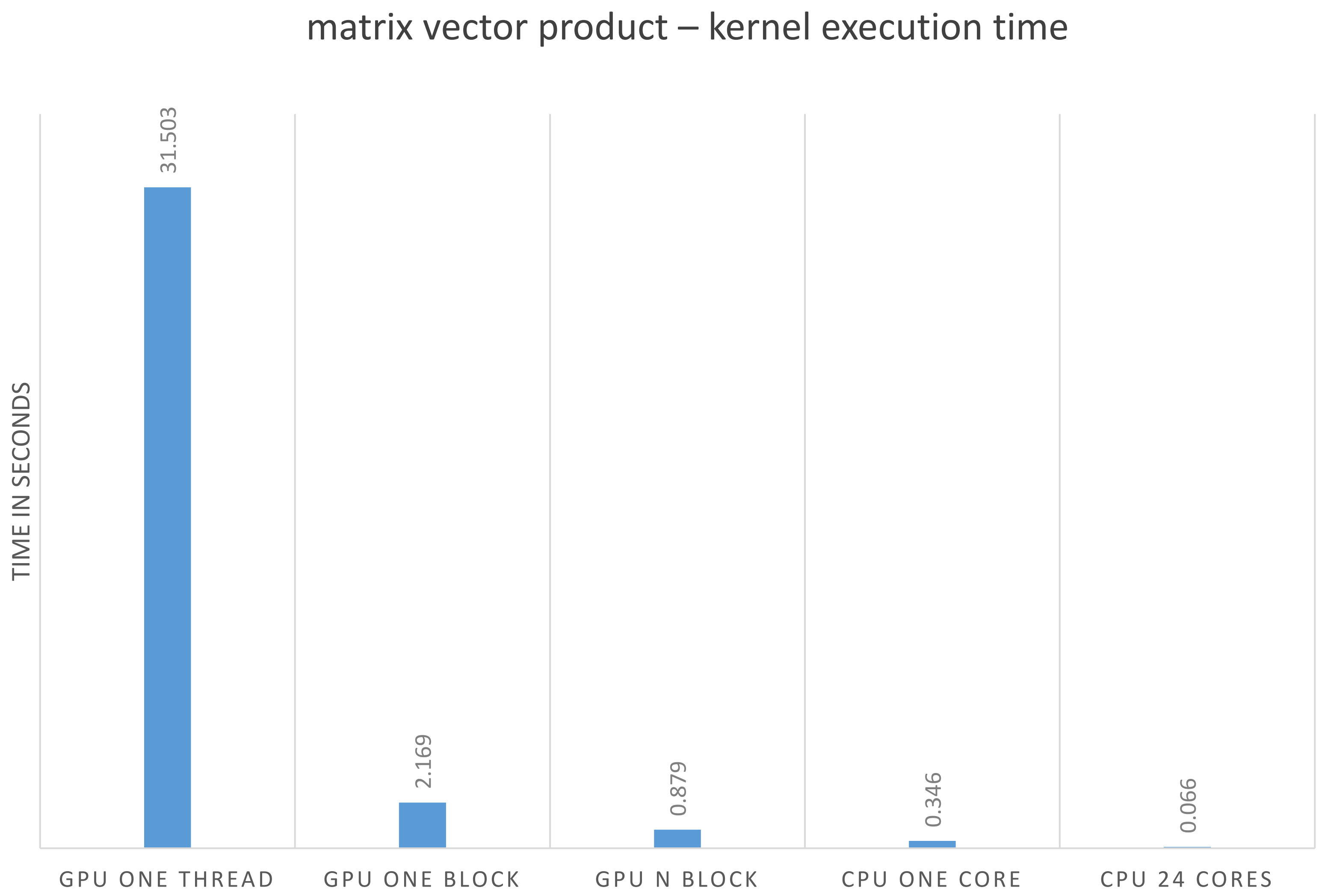
To measure time:

```
#include <sys/time.h>
double mysecond()
{
    struct timeval tp;
    struct timezone tzp;
    int i;

    i = gettimeofday(&tp,&tzp);
    return ( (double) tp.tv_sec + (double) tp.tv_usec  * 1.e-6);
}
```

```
t = mysecond();
                                        .
                                        .
                                        .
t = (mysecond() - t);
printf("Elapsed time = %g", t);
```

Time taken to copy data to device, perform computation, and copy from host is measured

Time comparison



Remark

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Recall that:

When using GPU

- Most of the time will be spent on copying data between CPU and GPU memory.
- GPU is ideal when many computations need to be done for a given data.

Iterative Solver for FEM

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Iterative solver for $Ax = b$

- Conjugate Gradient
- GMRES
- Orthomin
- Orthores
- Orthodir
- Bi Conjugate Gradient

Conjugate Gradient Method

$r_0 = b - Ax_0$
$p_0 = r_0$
$k = 0$
if $r^{\mathrm{T}} r <$ tol **then**
  $\alpha_k = \frac{r_k^{\mathrm{T}} r_k}{p_k^{\mathrm{T}} A p_k}$
  $x_{k+1} = x_k + \alpha_k p_k$
  $r_{k+1} = r_k + \alpha_k A p_k$
  $\beta_k = \frac{r_{k+1}^{\mathrm{T}} r_{k+1}}{p_k^{\mathrm{T}} A p_k}$
  $p_{k+1} = r_{k+1} + \beta_k p_k$
  $k = k + 1$
end if
return x_{k+1} as the result

In one iteration

- ❑ One Matrix - Vector Product
- ❑ Two vector dot product
- ❑ Four vectors of working stage

Iterative Solvers for FEM

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

In FEM:

- The system matrix K is assembled
- The iterative solver is used to find u for given f

How does iterative solver work?

1. An initial arbitrary guess of the solution is made for u
2. The solver starts a loop
3. In each iteration of the loop
    - Perform a matrix-vector multiplication Ku
    - Compute variables α, β, and p
    - u is updated
    - The residue $r = f - Ku$ decreases
4. When the residue r becomes less than an acceptable tolerance, the solver exits

Conjugate Gradient Method

$r_0 = b - Ax_0$

$p_0 = r_0$

$k = 0$

if $r^{\mathrm{T}} r <$ tol **then**

$\quad \alpha_k = \frac{r_k^{\mathrm{T}} r_k}{p_k^{\mathrm{T}} A p_k}$

$\quad x_{k+1} = x_k + \alpha_k p_k$

$\quad r_{k+1} = r_k + \alpha_k A p_k$

$\quad \beta_k = \frac{r_{k+1}^{\mathrm{T}} r_{k+1}}{p_k^{\mathrm{T}} A p_k}$

$\quad p_{k+1} = r_{k+1} + \beta_k p_k$

$\quad k = k + 1$

end if

return x_{k+1} as the result

Iterative Solvers for FEM

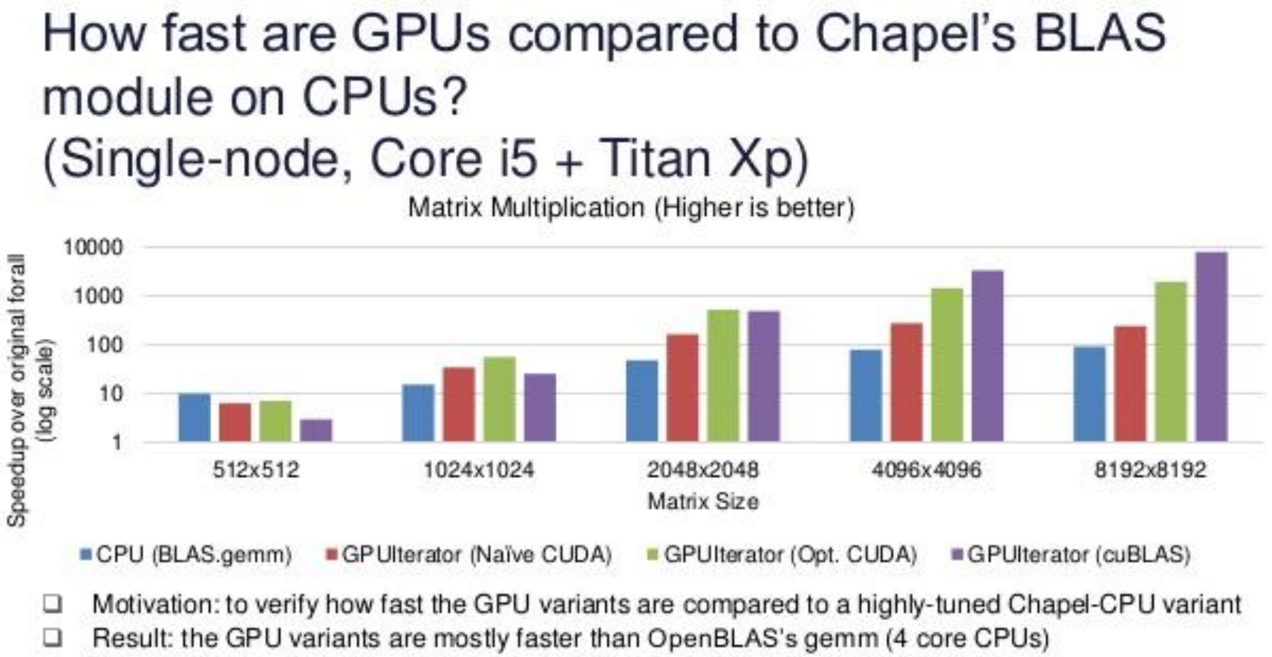
Conjugate Gradient Method

$r_0 = b - Ax_0$
$p_0 = r_0$
$k = 0$
if $r^{\mathrm{T}}r <$ tol **then**
  $\alpha_k = \frac{r_k^{\mathrm{T}} r_k}{p_k^{\mathrm{T}} A p_k}$
  $x_{k+1} = x_k + \alpha_k p_k$
  $r_{k+1} = r_k + \alpha_k A p_k$
  $\beta_k = \frac{r_{k+1}^{\mathrm{T}} r_{k+1}}{p_k^{\mathrm{T}} A p_k}$
  $p_{k+1} = r_{k+1} + \beta_k p_k$
  $k = k + 1$
end if
return x_{k+1} as the result

- The matrix A, the vectors b, and x_0 are copied to the GPU memory only once.
- The loop runs until the given tol is satisfied.
- The vectors change at each iteration. But they remain in GPU memory.
- The matrix A does not change.
- Many matrix-vector product are carried with the same matrix.
- This is one example where GPU can be efficient.

BLAS

- The BLAS (Basic Linear Algebra Subprograms) are routines for performing basic vector and matrix operations.
- Use library that utilises BLAS and compiler directives instead of programming.



The ACM SIGPLAN 6th Annual Chapel Implementers and Users Workshop (CHIUW2019) co-located with PLDI 2019 / ACM FCRC 2019.
https://www.slideshare.net/ahayashi10/gpuiterator-bridging-the-gap-between-chapel-and-gpu-platforms

Additional resources

Matrix Vector product using GPU CUBLAS program

FOLDER: CUDABLAS

https://developer.nvidia.com/sites/default/files/akamai/cuda/files/Misc/mygpu.pdf

Thank you for your attention!

http://sctrain.eu/

Univerza v Ljubljani
TECHNISCHE UNIVERSITÄT WIEN
CINECA consorzio interuniversitario
VSB TECHNICAL UNIVERSITY OF OSTRAVA | IT4INNOVATIONS NATIONAL SUPERCOMPUTING CENTER

Co-funded by the Erasmus+ Programme of the European Union

This project has been funded with support from the European Commission.
This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.