LOOP PARALLELIZATION

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

The material in these slides have been taken from Section 11.5-11.7 of the "Dragon Book"

Automatic Parallelization

- One of the Holy Grails of compiler research is automatic parallelization.
- We say that a program is *sequential* if developers did not take parallelism into consideration when coding it.
- We say that a machine is parallel if it can run several threads or processes at the same time.
  - Henceforth, we will be only interested in running several threads at the same time.

THE HOLY QUESTION: HOW TO COMPILE A SEQUENTIAL PROGRAM TO A PARALLEL MACHINE, SO THAT IT TAKES MAXIMUM BENEFIT FROM PARALLELISM?

What is the difference between thread and process?

The Gold is in the Loops

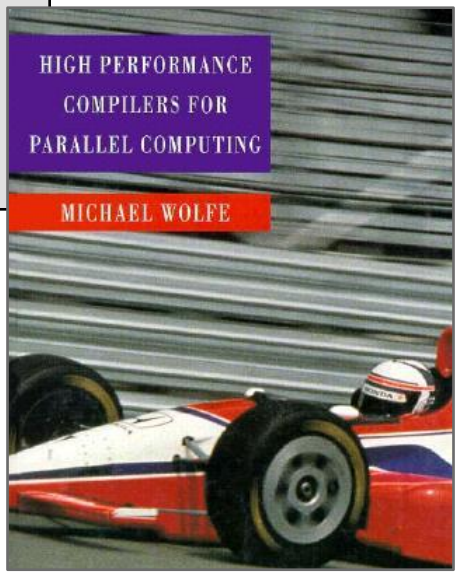
- Most parallelizing compilers focus on loops.
- Some loops can be translated to highly parallel code.
  - Others cannot.

1
```
for (i = 0; i < N; i++) {
    a[i] = 0;
}
```

2
```
for (i = N-2; i ≥ 0; i++) {
    a[i] = a[i] - a[i + 1];
}
```

1) Which of these two loops is more parallelizable?

2) In general, which factors determine when a loop is more or less parallelizable?



Loops are so important that there are even whole books that only talk about how to generate good code for them.

GPUs as a Target

- In the good old days, it was hard to find a massively parallel hardware at an accessible cost.
- Fortunately, these days are good and gone.
  - Modern GPUs gives us hundreds of cores for a few bucks.
- We shall be seeing some techniques to translate C loops to kernels implemented in C for CUDA♤.

The Nvidia GTX 570 can run 1,536 threads at the same time, on 14 Stream Multiprocessors clocked at 1,464MHz!

Which factors must be taken into consideration when deciding if we implement an algorithm to the GPU or to the CPU?

♤: We are assuming that you already know a bit about GPUs. If you don't, then take a look into our class about *Divergence Analysis*.

When to Run Code on the GPU?

```
void matMulCPU (float* B, float* C, float* A, unsigned int Width) {
  for (unsigned int i = 0; i < Width; ++i) {
    for (unsigned int j = 0; j < Width; ++j) {
      A[i * Width + j] = 0.0;
      for (unsigned int k = 0; k < Width; ++k) {
        A[i * Width + j] += B[i * Width + k] * C[k * Width + j];
      }
    }
  }
}
```

```
void matSumCPU(float* B, float* C, float* A, unsigned int Width) {
  for (unsigned int i = 0; i < Width; ++i) {
    for (unsigned int j = 0; j < Width; ++j) {
      A[i * Width + j] = B[i * Width + j] + C[i * Width + j];
    }
  }
}
```

Which of these two algorithm would benefit more from the parallelism available in the GPU?

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

The Long and Perilous Trip

- If we want to run a program on the GPU, we need to move the data that this program manipulates from the host (the CPU) to the device (the GPU).
  - Copying this data is pretty expensive.
- If our parallel complexity is not much better than the sequential complexity, then the cost of moving the data back and forth makes the use of the GPU prohibitive.
- In other words, it is worthwhile to run a program on the GPU when:

$$O^p + O^d \leq O^s$$

O^p = Parallel Complexity

O^d = Communication Complexity

O^s = Sequential Complexity

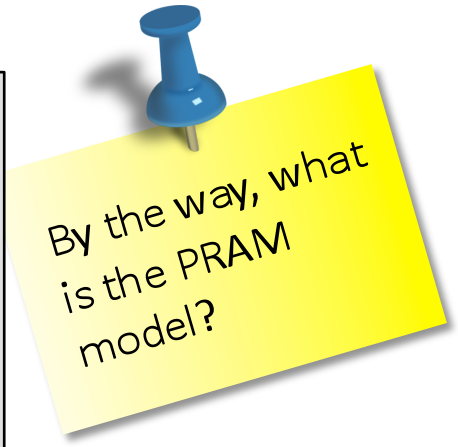
Matrix Multiplication in C

```
void matMulCPU(float* B, float* C, float* A, unsigned int Width) {
  for (unsigned int i = 0; i < Width; ++i) {
    for (unsigned int j = 0; j < Width; ++j) {
      A[i * Width + j] = 0.0;
      for (unsigned int k = 0; k < Width; ++k) {
        A[i * Width + j] += B[i * Width + k] * C[k * Width + j];
      }
    }
  }
}
```

1) What is the complexity of this code on a sequential machine?

2) What is the parallel complexity of this code, in the **PRAM** model?

3) What is the cost of moving data to and from the GPU?



Matrix Multiplication in C

```
void matMulCPU(float* B, float* C, float* A, unsigned int Width) {
  for (unsigned int i = 0; i < Width; ++i) {
    for (unsigned int j = 0; j < Width; ++j) {
      A[i * Width + j] = 0.0;
      for (unsigned int k = 0; k < Width; ++k) {
        A[i * Width + j] += B[i * Width + k] * C[k * Width + j];
      }
    }
  }
}
```

In case you feel like running this code yourself, to plot a chart, you can try **this** python driver.

```
import time
import subprocess

for M in range(1000, 3000, 500):
  for N in range(1000, 3000, 500):
    start = time.time()
    subprocess.call(['./mulMatrix', str(M), str(N)])
    elapsed = (time.time() - start)
    size = M * N
    print (str(size) + ', ' + str(elapsed))
```

Matrix Multiplication in CUDA

```
__global__ void matMulCUDA(float* B, float* C, float* A, int Width) {
  float Pvalue = 0.0;

  int tx = blockIdx.x * blockDim.x + threadIdx.x;
  int ty = blockIdx.y * blockDim.y + threadIdx.y;

  for (int k = 0; k < Width; ++k) {
    Pvalue += B[ty * Width + k] * C[k * Width + tx];
  }

  A[tx + ty * Width] = Pvalue;
}
```

1) What is the complexity of this version of matrix multiplication in C for CUDA?

2) What happened with all those loops in our original algorithm?

3) How many threads will run this program?

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

To run or not to run…

Matrix multiplication is a typical application that benefits from all that parallelism available on the GPU. The calculation that we do is like this:

It is worth using the GPU when: $O^p + O^d \leq O^s$

In the case of matrix multiplication, we have:

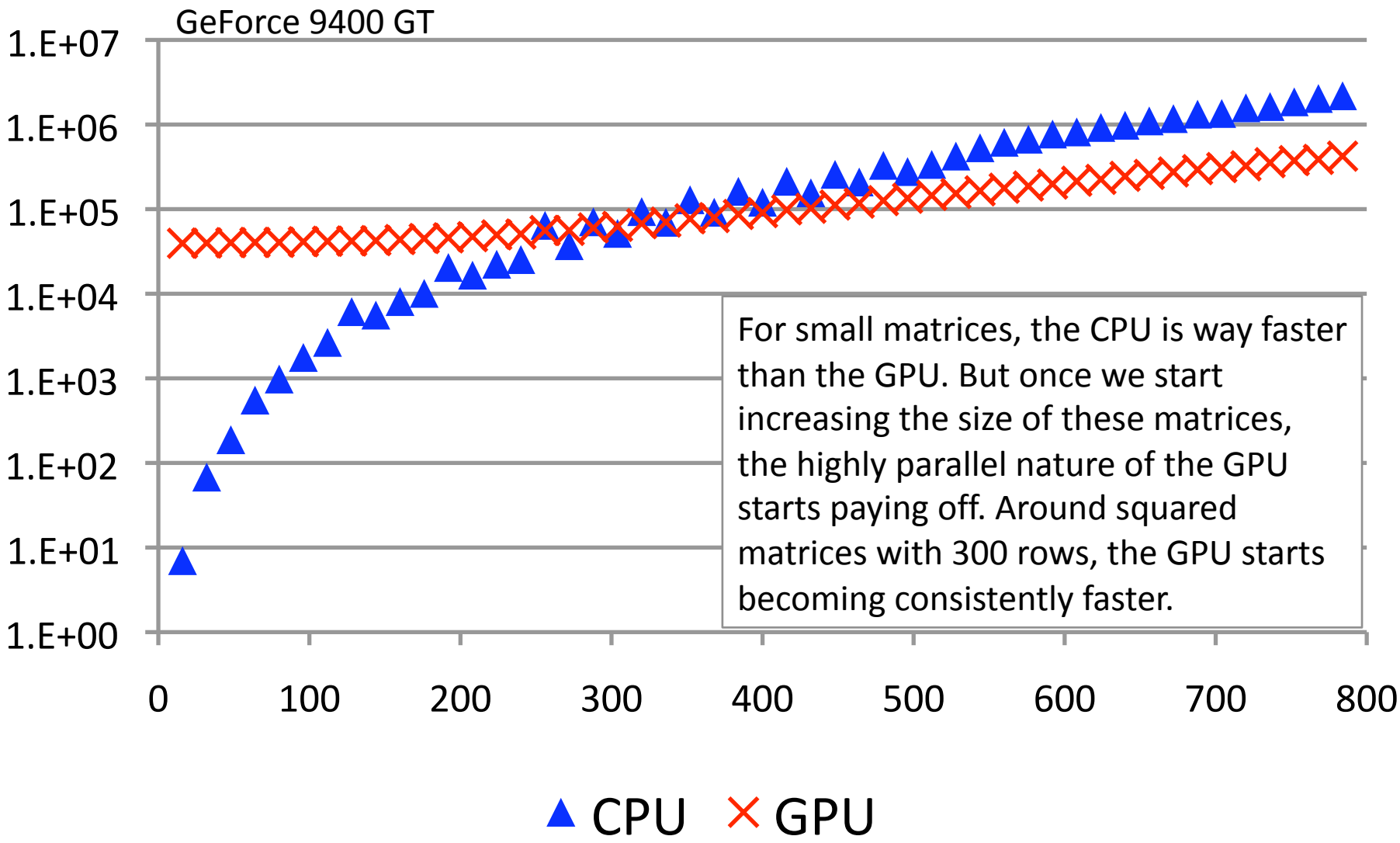
O^p = Parallel Complexity $\Rightarrow O(N)$

O^d = Communication Complexity $\Rightarrow O(N^2)$

O^s = Sequential Complexity $\Rightarrow O(N^3)$

So, in the end, we have $O(N) + O(N^2) = O(N^2) \leq O(N^3)$

Matrix Multiplication: CPU × GPU



Matrix Sum in the CPU

```
void matSumCPU(float* B, float* C, float* A, unsigned int Width) {
  for (unsigned int i = 0; i < Width; ++i) {
    for (unsigned int j = 0; j < Width; ++j) {
      A[i * Width + j] = B[i * Width + j] + C[i * Width + j];
    }
  }
}
```

$$\begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix} + \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} = \begin{bmatrix} a_1 + a_2 & b_1 + b_2 \\ c_1 + c_2 & d_1 + d_2 \end{bmatrix}$$

1) What is the complexity of this code on a sequential machine?

2) What is the parallel complexity of this code, in the PRAM model?

3) What is the cost of moving data to and from the GPU?

Matrix Sum in the GPU

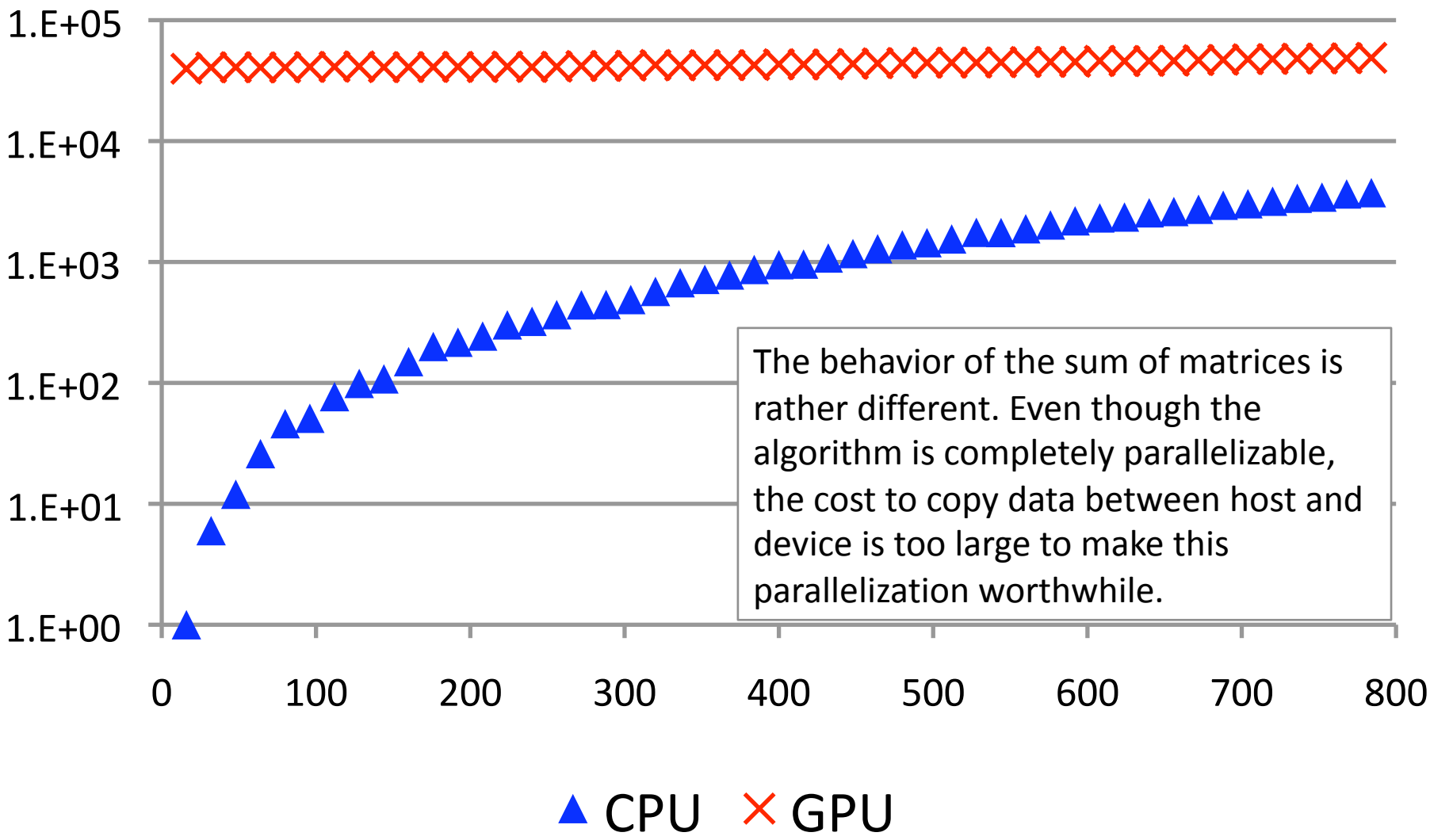
```
__global__ void matSumCUDA(float* B, float* C, float* A, int Width) {
  int tx = blockIdx.x * blockDim.x + threadIdx.x;
  int ty = blockIdx.y * blockDim.y + threadIdx.y;
  A[tx + ty * Width] = B[tx + ty * Width] + C[tx + ty * Width];
}
```

1) What is the complexity of this version of matrix multiplication in C for CUDA?

2) How many threads will run this program?

3) So, in the end, is it worthwhile running matrix sum on the GPU?

Matrix Sum: CPU × GPU



To run or not to run… the same old decision

Matrix sum does not benefit too much from all that parallelism available on the GPU. Again, remember that it is worth using the GPU when: $O^p + O^d \leq O^s$

In the case of matrix sum, we have:

O^p = Parallel Complexity $\Rightarrow O(N)$

O^d = Communication Complexity $\Rightarrow O(N^2)$

O^s = Sequential Complexity $\Rightarrow O(N^2)$

So, in the end, we have $O(N) + O(N^2) = O(N^2) \not\leq O(N^2)$

Making Decisions

- In general, it is worth running a program on the GPU when:
  - This program has available parallelism.
  - The data tends to be *reused* often.
- There are techniques that we can use to estimate the amount of parallelism in a loop, and the amount of reuse of the data.
- These techniques use a bit of linear algebra.
  - Oh, you did not think you would be rid of it, did you?

1) How to estimate how many times each position of an array is used inside a loop?
2) How to estimate the amount of parallelism within the loop?

Data Reuse

- Data reuse is a measure of how often data is accessed within a loop.
- If the data is just used once, then it may not be worthwhile to send it to the GPU, for instance.

```
for (i = 1; i < N; i++)
    for (j = 0; j < N – 1; j++)
        X[i – 1] = …;
        Y0[i, j] = …;
        Y1[j, j + 1] = …;
        Y2[1, 2] = …;
        Z[1, i, 2*i + j] = …;
```

How many times each position of each array is written within this loop?

Data Reuse

- Data reuse is a measure of how often data is accessed within a loop.

Can you come up with a general rule?

```
for (i = 1; i < N; i++)
    for (j = 0; j < N – 1; j++)
        X[i – 1] = …;
        Y0[i, j] = …;
        Y1[j, j + 1] = …;
        Y2[1, 2] = …;
        Z[1, i, 2*i + j] = …;
```

Each X[i - 1] is written once for each different j, so this location is reused O(N) times
Each $Y_0[i, j]$ is written once for each pair i, j, so this data is reused only once.
Each $Y_1[j, j+1]$ is accessed once for each i, so this location is reused O(N) times.
$Y_2[1, 2]$ is written at each iteration of the loop, so this location is reused O(N×N) = $O(N^2)$ times.
Each Z[1, i, 2*i + j] is written once for each possible combination of i and j, so these locations are only reused once apiece.

Comparing Dimensions

- As a first approximation, if we access a k-dimensional array within d nested loops, each one with an O(N) trip count, then we tend to reuse the same data $O(N^{d-k})$ times.

```
for (i = 1; i < N; i++)
  for (j = 0; j < N – 1; j++)
    X[i – 1] = …;
    Y0[i, j] = …;
    Y1[j, j + 1] = …;
    Y2[1, 2] = …;
    Z[1, i, 2*i + j] = …;
```

- This rule works in these two first iterations, but it does not quite work on the last three.

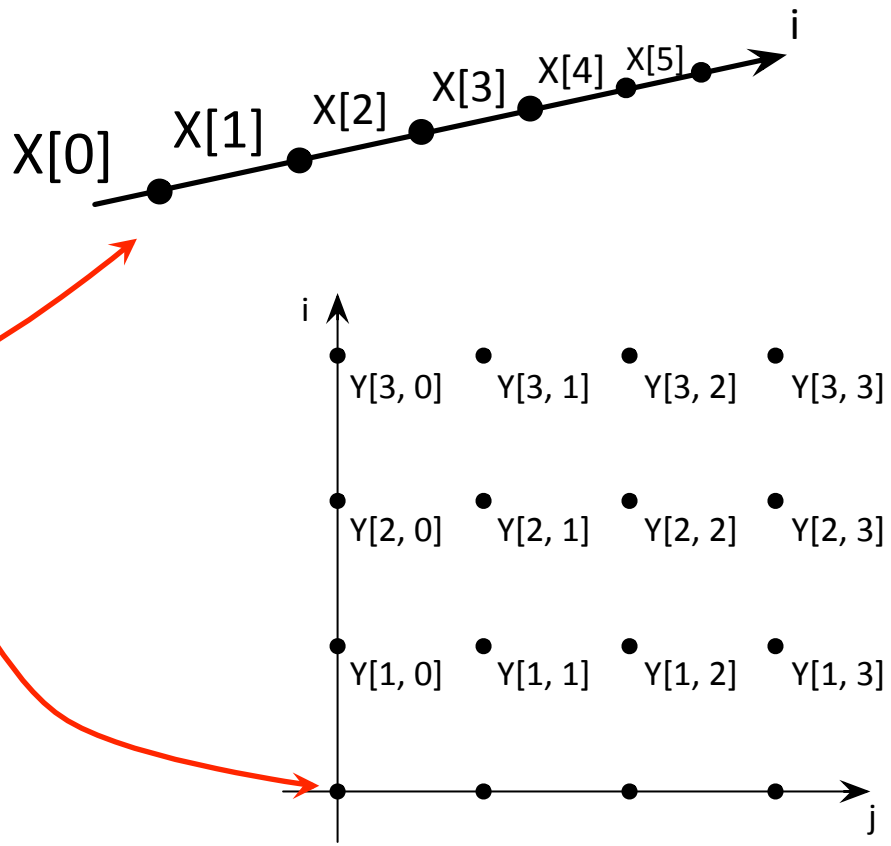
Can you come up with a way to estimate the "dimension" of each of these three last accesses?

The Data Access Space

- The data access space is a collection of points that determine which indices of an array are accessed within a loop.

How to find the dimension of the data space?

```
for (i = 1; i < N; i++)
   for (j = 0; j < N – 1; j++)
      X[i – 1] = ...;
      Y0[i, j] = ...;
      Y1[j, j + 1] = ...;
      Y2[1, 2] = ...;
      Z[1, i, 2*i + j] = ...;
```



Accesses as Matrices

- We can represent an array access such as X[ai + bj + c, di + ej + f] using a matrix notation.

$$\begin{bmatrix} a & b \\ d & e \end{bmatrix} \times \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} c \\ f \end{bmatrix}$$

```
1 for (i = 0; i < N; ++i) {
2   for (j = 0; j < N; ++j) {
3     A[i, j] = 0.0;
4     for (k = 0; k < N; ++k) {
5       A[i, j] += B[i, k] * C[k, j];
6     }
7   }
8 }
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Accesses as Matrices

- We can represent an array access such as X[ai + bj + c, di + ej + f] using a matrix notation.

$$\begin{bmatrix} a & b \\ d & e \end{bmatrix} \times \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} c \\ f \end{bmatrix}$$

```
for (i = 1; i < N; i++)
    for (j = 0; j < N – 1; j++)
        X[i – 1] = …;
        Y0[i, j] = …;
        Y1[j, j + 1] = …;
        Y2[1, 2] = …;
        Z[1, i, 2*i + j] = …;
```

What is the matrix representation of the data space of each of these array accesses?

Accesses as Matrices

The dimension of the data access space is the rank of the *coefficient matrix* F, in $F[i\ j]^t + f$

```
for (i = 1; i < N; i++)
  for (j = 0; j < N – 1; j++)
    X[i – 1] = ...;
    Y0[i, j] = ...;
    Y1[j, j + 1] = ...;
    Y2[1, 2] = ...;
    Z[1, i, 2*i + j] = ...;
```

$$\begin{bmatrix}1 & 0\end{bmatrix} \times \begin{bmatrix}i\\ j\end{bmatrix} + \begin{bmatrix}-1\end{bmatrix}$$

$$\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} \times \begin{bmatrix}i\\ j\end{bmatrix} + \begin{bmatrix}0\\ 0\end{bmatrix}$$

$$\begin{bmatrix}0 & 1\\ 0 & 1\end{bmatrix} \times \begin{bmatrix}i\\ j\end{bmatrix} + \begin{bmatrix}0\\ 1\end{bmatrix}$$

$$\begin{bmatrix}0 & 0\\ 0 & 0\end{bmatrix} \times \begin{bmatrix}i\\ j\end{bmatrix} + \begin{bmatrix}1\\ 2\end{bmatrix}$$

$$\begin{bmatrix}0 & 0\\ 1 & 0\\ 2 & 1\end{bmatrix} \times \begin{bmatrix}i\\ j\end{bmatrix} + \begin{bmatrix}1\\ 0\\ 0\end{bmatrix}$$

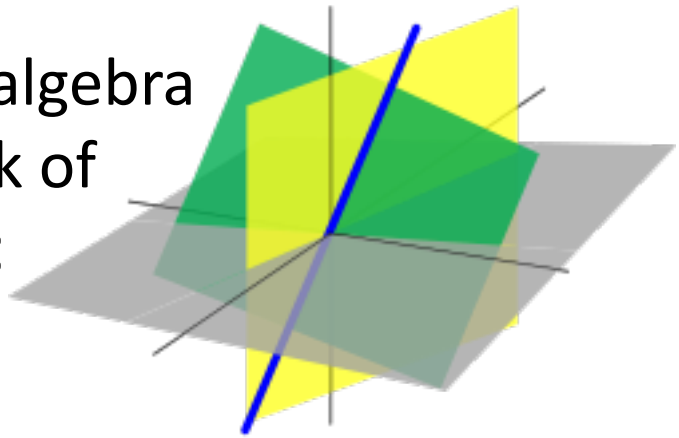
Do you remember what is the "rank" of a matrix?

The Rank of a Matrix

- The rank of a matrix F is the largest number of columns (or rows) of F that are linearly independent.
- A set of vectors is linearly independent if none of the vectors can be written as a linear combination of the others.

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

- For instance, the rank of this matrix on the left is 2, because the first row, e.g., R_0, can be written as $0 \times R_1 + 0 \times R_2$.
- There are standard linear algebra techniques to find the rank of matrices, and we shall **not** review them here.



Estimating Reuse from the Rank of Data Space

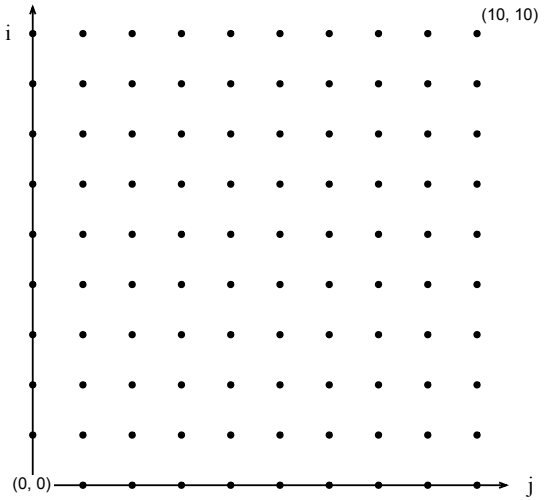
- If the *iteration space* of a loop has d dimensions, each one with an O(N) trip count, then an access whose data space has rank k will be reused $O(N^{d-k})$ times.
- That is the first time that we talk about iteration space of a loop, but the concept is straightforward:
  - The iteration space of a nested loop is the set of possible combinations of trip counts of every loop in the nest.

```
for (unsigned int i = 0; i < N; ++i)
   for (unsigned int j = 0; j < N; ++j)
      A[i, j] = B[i, j] + C[i, j];
```

What is the iteration space of our matSum example?

Iteration Spaces

```
for (unsigned int i = 0; i < N; ++i)
   for (unsigned int j = 0; j < N; ++j)
      A[i, j] = B[i, j] + C[i, j];
```

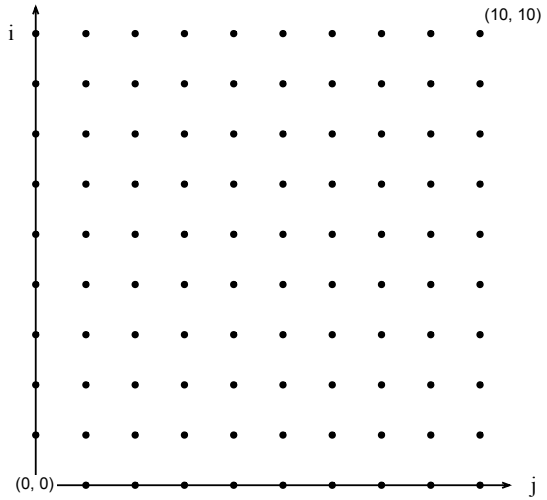



```
for (unsigned int i = 0; i < N; ++i)
   for (unsigned int j = 0; j < N; ++j) {
      A[i, j] = .0;
      for (unsigned int k = 0; k < N; ++k) {
         A[i, j] += B[i, k] * C[k, j];
   }
```

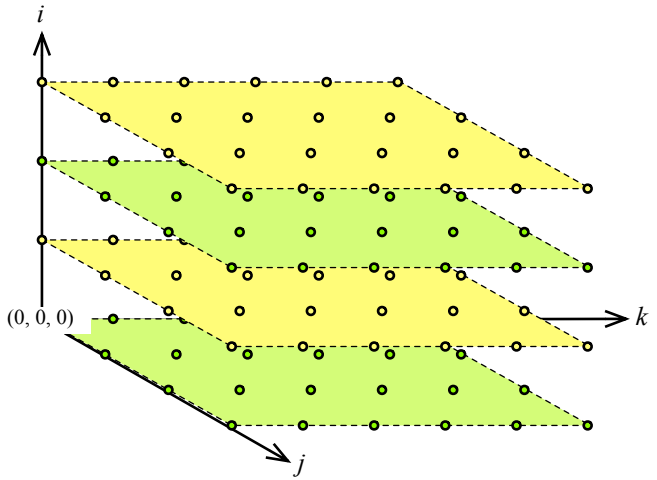
And how is the iteration space of our mulMat example?

Iteration Spaces

```
for (unsigned int i = 0; i < N; ++i)
    for (unsigned int j = 0; j < N; ++j)
        A[i, j] = B[i, j] + C[i, j];
```




```
for (unsigned int i = 0; i < N; ++i)
    for (unsigned int j = 0; j < N; ++j) {
        A[i, j] = .0;
        for (unsigned int k = 0; k < N; ++k) {
            A[i, j] += B[i, k] * C[k, j];
    }
```



The Pigeonhole Principle

- if *n* items are put into *m* pigeonholes with *n* > *m*, then at least one pigeonhole must contain more than one item.

In this figure, taken from wikipedia, we have *n* = 10 pigeons in *m* = 9 holes. Since 10 is greater than 9, the pigeonhole principle says that at least one hole has more than one pigeon.

If the iteration space has more points than the data access space and data is accessed at each iteration, then some points in the data space will be reused in different iterations, by the pigeonhole principle.

In general it is not so easy to tell the exact number of points in either the data space or in the iteration space. We approximate these notions by using the rank, i.e., the number of dimensions of each kind of space.

Estimating Reuse

X[i-1]

$$[1 \quad 0] \times \begin{bmatrix} i \\ j \end{bmatrix} + [-1]$$

$Y_0[i, j]$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$Y_1[j, j+1]$

$$\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$Y_2[1, 2]$

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \times \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Z[1, i, 2 * i + j]

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \times \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Remember, reuse can be approximated by $O(N^{d-k})$, where d is the rank of the iteration space, k is the rank of the data space, and N is the trip count of each iteration.

1) What is the rank of each of the coefficient matrices seen on the left?

2) Assuming a 2D iteration space, what is the amount of reuse available in each example?

Estimating Reuse

	Access:	Rank:	Reuse:
$\begin{bmatrix}1 & 0\end{bmatrix} \times \begin{bmatrix}i\\ j\end{bmatrix} + \begin{bmatrix}-1\end{bmatrix}$	X[i-1]	1	O(N)
$\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} \times \begin{bmatrix}i\\ j\end{bmatrix} + \begin{bmatrix}0\\ 0\end{bmatrix}$	Y_0[i, j]	2	O(1)
$\begin{bmatrix}0 & 1\\ 0 & 1\end{bmatrix} \times \begin{bmatrix}i\\ j\end{bmatrix} + \begin{bmatrix}0\\ 1\end{bmatrix}$	Y_1[j, j+1]	1	O(N)
$\begin{bmatrix}0 & 0\\ 0 & 0\end{bmatrix} \times \begin{bmatrix}i\\ j\end{bmatrix} + \begin{bmatrix}1\\ 2\end{bmatrix}$	Y_2[1, 2]	0	$O(N^2)$
$\begin{bmatrix}0 & 0\\ 1 & 0\\ 2 & 1\end{bmatrix} \times \begin{bmatrix}i\\ j\end{bmatrix} + \begin{bmatrix}1\\ 0\\ 0\end{bmatrix}$	Z[1, i, 2 * i + j]	2	O(1)

Reuse in Matrix Sum

```
for (unsigned int i = 0; i < N; ++i)
    for (unsigned int j = 0; j < N; ++j)
        A[i, j] = B[i, j] + C[i, j];
```

1) Can you write each array access in matrix notation?
2) What is the rank of each matrix?
3) How much reuse do we have in this loop?

Reuse in Matrix Sum

```
for (unsigned int i = 0; i < N; ++i)
    for (unsigned int j = 0; j < N; ++j)
        A[i, j] = B[i, j] + C[i, j];
```

All the array accesses have this format:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The coefficient matrix has rank 2. This is the rank of the data access space. Given that our iteration space has rank 2, we have that each access is reused $O(N^{2-2})$ times. In other words, we read each memory position only once in this nested loop. Copying this data to the GPU, to use it only once might not be a good idea.

Reuse in Matrix Multiplication

```
for (unsigned int i = 0; i < N; ++i)
    for (unsigned int j = 0; j < N; ++j) {
        A[i, j] = .0;
        for (unsigned int k = 0; k < N; ++k)
            A[i, j] += B[i, k] * C[k, j];
    }
```

Same as before:

1) Can you write each array access in matrix notation?
2) What is the rank of each matrix?
3) How much reuse do we have in this loop?

Reuse in Matrix Multiplication

```
for (unsigned int i = 0; i < N; ++i)
    for (unsigned int j = 0; j < N; ++j) {
        A[i, j] = .0;
        for (unsigned int k = 0; k < N; ++k)
            A[i, j] += B[i, k] * C[k, j];
    }
```

Each coefficient matrix has rank 2. All the accesses occur within a 3D iteration space. Hence, each access is reused $O(N^{3-2}) = O(N)$ times.

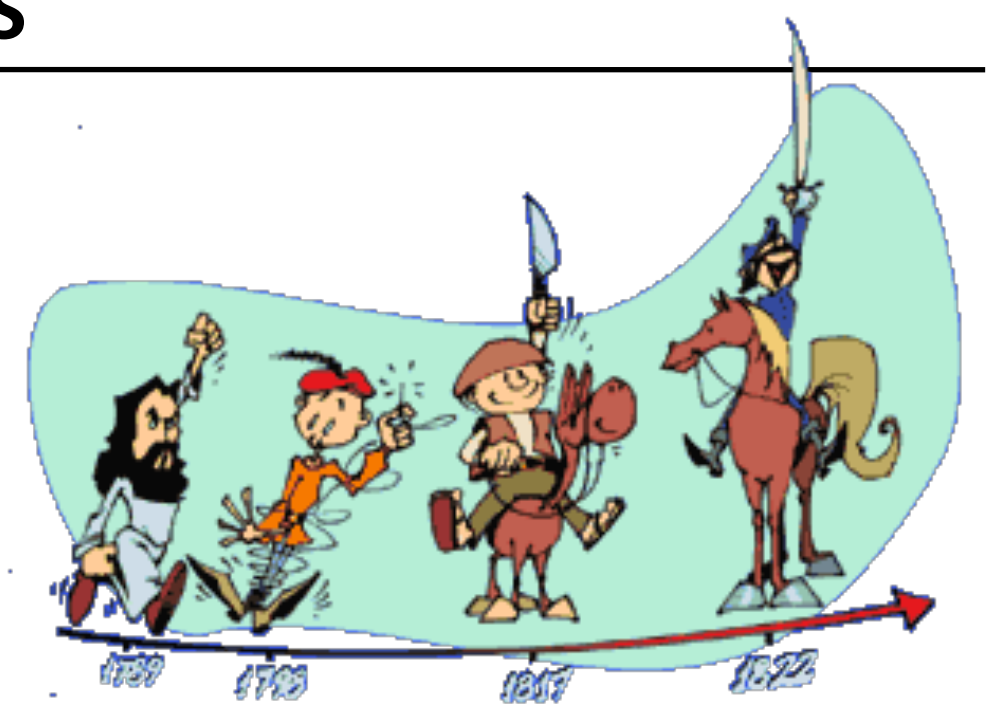
$$\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix}\times\begin{bmatrix}i\\ j\\ k\end{bmatrix}+\begin{bmatrix}0\\ 0\end{bmatrix} \quad \begin{bmatrix}1 & 0 & 0\\ 0 & 0 & 1\end{bmatrix}\times\begin{bmatrix}i\\ j\\ k\end{bmatrix}+\begin{bmatrix}0\\ 0\end{bmatrix} \quad \begin{bmatrix}0 & 0 & 1\\ 0 & 1 & 0\end{bmatrix}\times\begin{bmatrix}i\\ j\\ k\end{bmatrix}+\begin{bmatrix}0\\ 0\end{bmatrix}$$

Matrix Multiplication capitalizes well on all the power of the GPU. Each memory location is accessed many times. Thus, the computations end up paying for the high communication cost.

Compilers Laboratory

Department of Computer Science
Universidade Federal de Minas Gerais
Federal University of Minas Gerais, Brazil

Data Dependences



Data Dependences

- If two statements are independent, then we can process them in parallel.
- There exists a dependence between statements s_1 and s_2 if they access the same location, and at least one of these accesses is a write.
- Usually, when determining dependences between array accesses we need alias information, as we can see in the two examples on the right.
  - Henceforth, we shall assume that no two arrays are aliases.

```
A = malloc(…);
B = malloc(…);
for (int i = 1; i < N – 1; i++) {
   A[i] = B[i – 1];
}
```

```
A = malloc(…);
B = A;
for (int i = 1; i < N – 1; i++) {
   A[i] = B[i – 1];
}
```

Warm Up Problem

We can store a set of 2D points in an one-dimensional array as follows:

```
N = _getNumPoints(&Points);
a = malloc(2 * sizeof(double) * N);
for (i = 0; i < N; i++) {
   a[2 * i] = _getPoint(&Points, i).x;
}
for (j = 0; j < N; j++) {
   a[2 * j + 1] = _getPoint(&Points, j).y;
}
```

1) Can we have dependences between statements from different iterations of the same loop?

2) Can we have dependences between statements within different loops?

Prove that there is no way to write the same position of array "a" twice, either on the same loop, or in different loops. That is, prove that there exists no dependences between each array access a[…].

Warm Up Problem

We can store a set of 2D points in an one-dimensional array as follows:

```
N = _getNumPoints(&Points);
a = malloc(2 * sizeof(double) * N);
for (i = 0; i < N; i++) {
   a[2 * i] = _getPoint(&Points, i).x;
}
for (j = 0; j < N; j++) {
   a[2 * j + 1] = _getPoint(&Points, j).y;
}
```

1) Can we have dependences between statements from different iterations of the same loop?

2) Can we h
between
within dif

First we consider dependences between different executions of the same statement.

The first question asks us if it is possible to have 2 * i = 2 * i', where i ≠ i'. Well, trivially we have that 2 * (i – i') = 0 requires i = i'. So, the answer for the first question is **no**.

Warm Up Problem

We can store a set of 2D points in an one-dimensional array as follows:

```
N = _getNumPoints(&Points);
a = malloc(2 * sizeof(double) * N);
for (i = 0; i < N; i++) {
   a[2 * i] = _getPoint(&Points, i).x;
}
for (j = 0; j < N; j++) {
   a[2 * j + 1] = _getPoint(&Points, j).y;
}
```

Now we look into dependences between different statements.

e have dependences
en statements from
ent iterations of the
loop?

2) Can we have dependences between statements within different loops?

The second question is trickier: can we have 2 * i = 2 * j + 1, if i and j are integers? Well, 2 * (i – j) = 1; thus, i – j = 1 / 2. We cannot have a fractional difference between two integers, and the answer for the second question is also **no**.

The Greatest Common Divisor (GCD) Test

- The equation that we just saw is called *Diophantine*.
- A Diophantine Equation stipulates that solutions must be integers.
  - A general procedure to solve any Diophantine Equation was one of the open problems posed by Hilbert in the beginning of the twentieth century.
    - Today we know that there is no such a method.
    - But we can know for sure when a linear Diophantine Equation has no solution:

The linear Diophantine Equation

$a_1x_1 + a_2x_2 + \ldots + a_nx_n = c$

has an integer solution for $x_1, x_2, \ldots, x_n$ if, and only if, the $GDC(a_1, a_2, \ldots, a_n)$ divides c.

Loops and Diophantine Equations

In order to find dependences within loops, we must solve Diophantine Equations:

```
N = _getNumPoints(&Points);
a = malloc(2 * sizeof(double) * N);
for (i = 0; i < N; i++) {
   a[2 * i] = _getPoint(&Points, i).x;
   a[2 * i + 1] = _getPoint(&Points, i).y;
}
```

To find dependences between two different executions of **this** statement, we must solve the equation 2 * i = 2 * i', which has only one solution, e.g, i = i'

And to find dependences between both array accesses, e.g., a[2*i] and a[2*i + 1], we must solve this equation: 2*i = 2*i' + 1, which we can rewrite as 2*i – 2*i' = 1. From the GCD test, we know that it has no solution. In other words, GCD(2, 2) = 2, but 2 does not divide 1.

Diophantus of Alexandria

Finding Dependences

- Finding dependences between statements within the same iteration of a loop is easy.
  - Use plain and simple reaching definition analysis.
- In order to find all the dependencies between different iterations of a loop, we must
  - for each array access a[E] that is a write operation:
    - For each other array access, a[E']:
      - Solve the Diophantine Equation E = E'

We call such dependencies **inter loop dependences**

Can you find all the dependences between different iterations of this loop on the right?

```
for (i = 0; i < N; i++)
    for (j = 1; j < N; j++)
        a[i, j] = (a[i, j] + a[i, j - 1]) / 2;
```

This loop is not as contrived as it seems. It is part of a smoothing routine, used in image processing.

Finding Dependences

```
for (i = 0; i < N; i++)
   for (j = 1; j < N; j++)
      a[i, j] = (a[i, j] + a[i, j - 1]) / 2;
```

We must consider dependences between a[i, j] and itself, and between a[i, j] and a[i, j – 1].

First, let's see if we can have any dependency between a[i, j] and itself in two different iterations:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} i' \\ j' \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The only way we can solve this system is to have $i = i'$ and $j = j'$. In other words, the only time a[i, j] is reused is within the same iteration.

Can you perceive a relation between data dependences and data reuse?

Reuse and Data Dependences

The notion of data reuse is closely related to the concept of data dependence. Data dependences exist only when we reuse memory locations. Notice that we have different kinds of dependencies, but only talk about one type of "reuse":

Can we have reuse without dependences?

- True dependence is implied by a read after a write, e.g.,
  for (i = 1; i < N; i++) { a[i] = …; … = E(a[i – 1]); }
- Anti-Dependence is implied by a write after a read, e.g.,
  for (i = 0; i < N; i++) { a[i] = ...; … = E(a[i + 1])}.
- Output dependences are created by reuse due to a write after a write, e.g.,
  for (i = 1; i < N; i++) { a[i] = ...; a[i – 1] = ...}.

```
for (i = 0; i < N; i++)
   for (j = 1; j < N; j++)
      a[i, j] = (a[i, j] + a[i, j - 1]) / 2;
```

1) Is there reuse between a[i, j] and a[i, j – 1]?
2) Which kind of dependence is this one?

Finding Dependences

```
for (i = 0; i < N; i++)
   for (j = 1; j < N; j++)
      a[i, j] = (a[i, j] + a[i, j - 1]) / 2;
```

We want to know if there exist two points in the iteration space, e.g., (i × j) and (i' × j'), such that a[i', j'] = a[i, j – 1].

In other words, we want to know if the value stored at a[i, j – 1] can be produced by another iteration of the loop. If that is the case, then we should consider assigning these two iterations to the same processor, when generating parallel code.

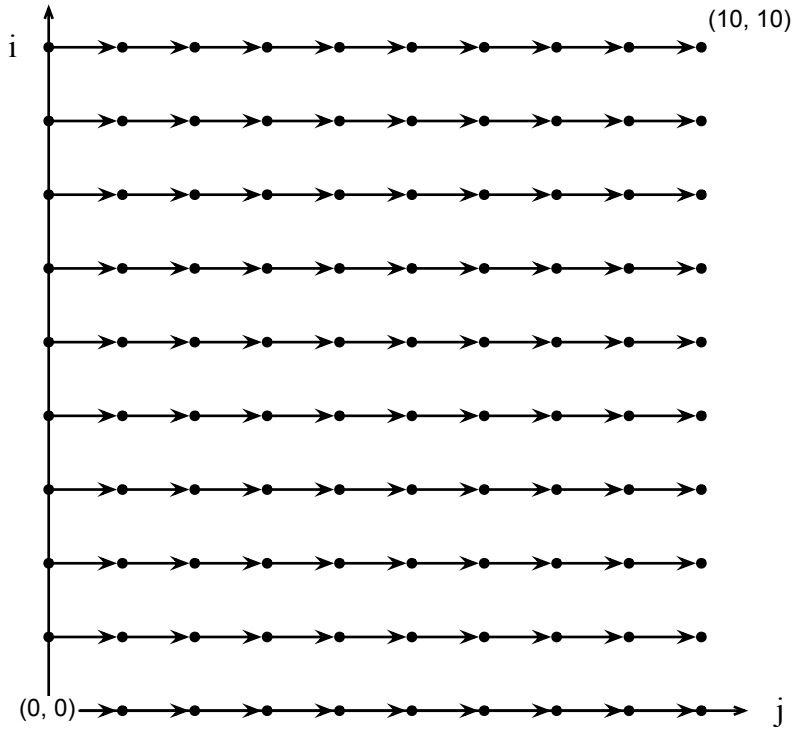
$$\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\times\begin{bmatrix}i\\ j\end{bmatrix}+\begin{bmatrix}0\\ -1\end{bmatrix}=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\times\begin{bmatrix}i'\\ j'\end{bmatrix}+\begin{bmatrix}0\\ 0\end{bmatrix}$$

This system gives us two Diophantine equations. The first is $i = i'$, which has only the trivial solution. The second is $j - 1 = j'$, which has solution for any j' such that $j' = j - 1$. In other words, on the j dimension, one iteration depends on the iteration that happened before.

How parallel is this example? How fast could we solve it on the PRAM model?

A Picture of Dependences

```
N = 10;
for (i = 0; i < N; i++)
   for (j = 1; j < N; j++)
      a[i, j] = (a[i, j] + a[i, j - 1]) / 2;
```



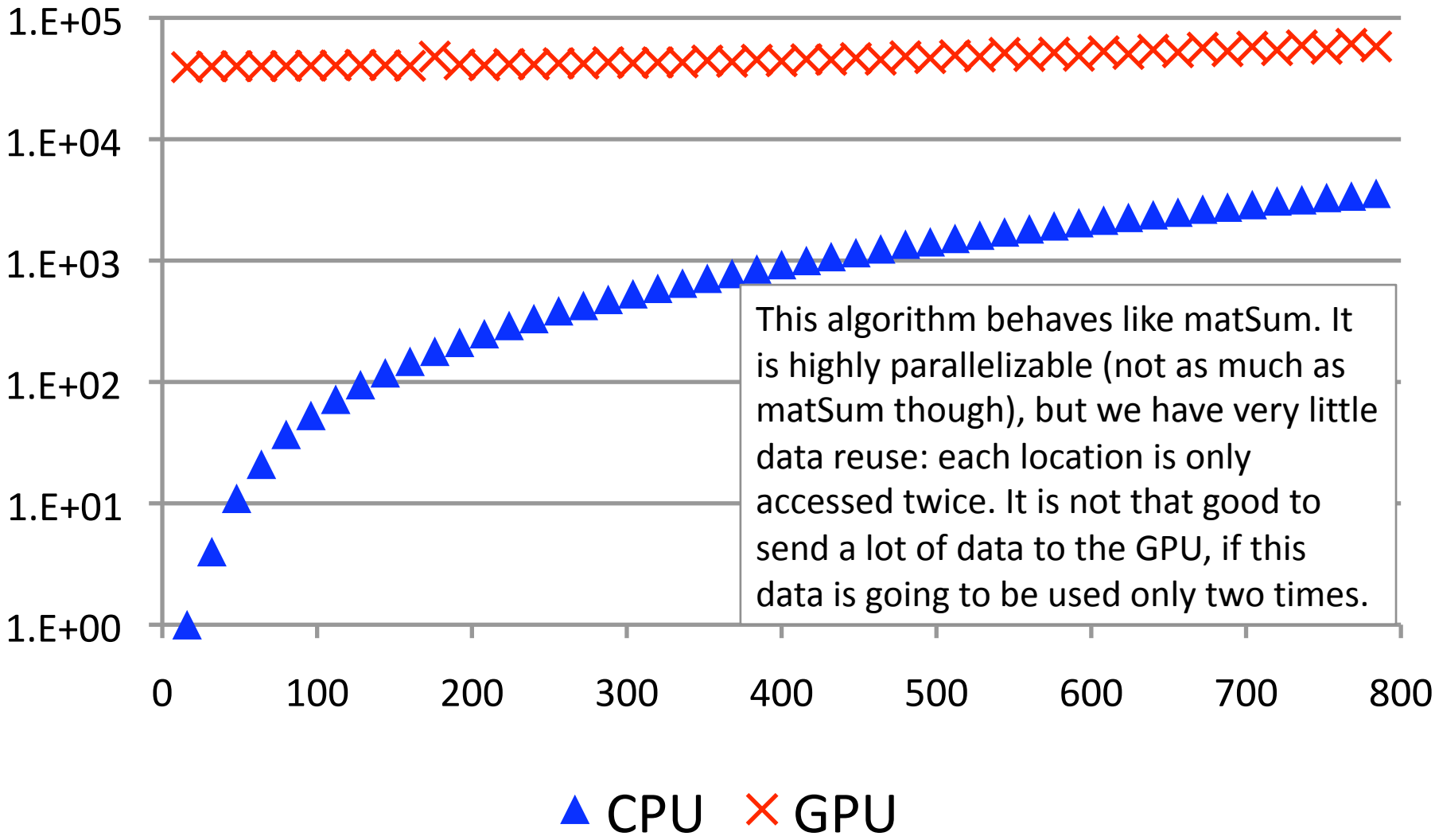
We can see below the iteration space of this loop. The arrows between different points denote dependences between different iterations of the loop. We can see lines on the *j* dimension. There are no dependences outside each line. Thus, we could assign each line to a different GPU thread, reducing from $O(N^2)$ to $O(N)$ the complexity of this loop.

```
__global__ void smoothing(float** a, int N) {
  int i = bid.x * bdim.x + tid.x;
  if (i < N)
    for (int j=1; j < N; j++) {
       a[i][j] = (a[i][j] + a[i][j – 1]) / 2;
    }
}
```

Is it worthwhile to run this program on the GPU?

```
for (i = 0; i < N; i++)
    for (j = 1; j < N; j++)
        a[i, j] = (a[i, j] + a[i, j - 1]) / 2;
```

Cost analysis



A Geometric View of Anti-Dependences

```
for (i = 0; i < N; i++)
   for (j = 1; j < N; j++)
      a[i, j] = (a[i, j] + a[i, j – 1]) / 2;
```

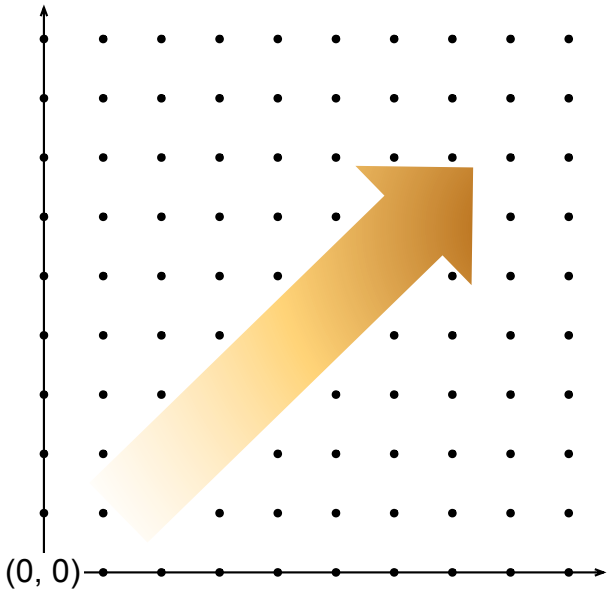
Let's consider a slightly modified view of our earlier example. Instead of iterating over a[i, j – 1], we shall consider a[i, j + 1].

```
for (i = 0; i < N; i++)
   for (j = 0; j < N-1; j++)
      a[i, j] = (a[i, j] + a[i, j + 1]) / 2;
```

This new example is curious in itself: iteration (i, j) reads a[i, j + 1]. Yet, this location, a[i, j + 1], will be only updated in a future iteration.

Well, an iteration cannot depend on something that has not happened yet. In other words, dependences in the iteration space go from the the null vector towards lexicographically larger vectors. In a 2D space, we would have:

Can you draw the pattern of dependences in the new kernel?

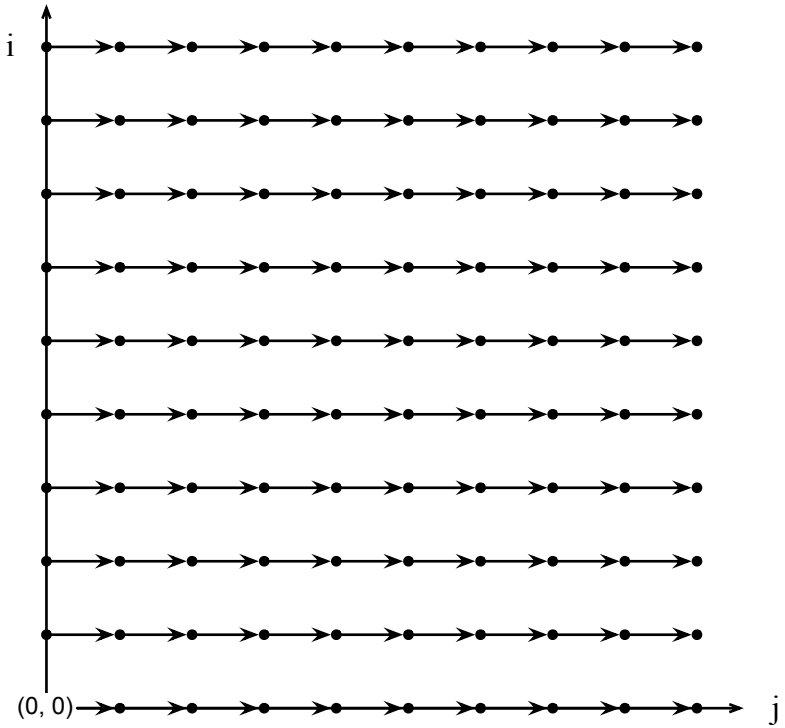


A Geometric View of Anti-Dependences

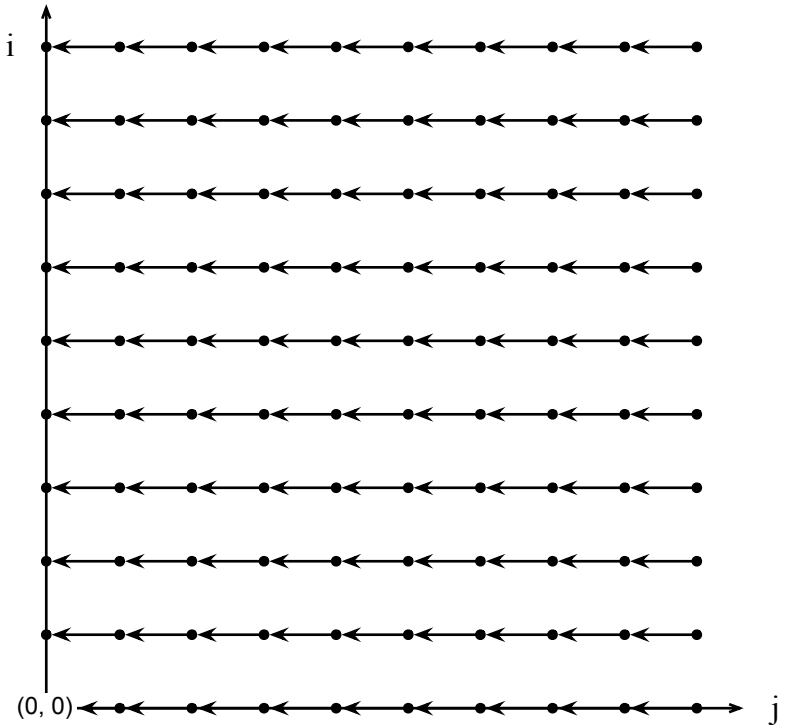
When we draw anti-dependences, the arrows in the iteration space appear in the opposite direction as the dependence arrows in the data space:

```
for (i = 0; i < N; i++)
    for (j = 0; j < N-1; j++)
        a[i, j] = (a[i, j] + a[i, j + 1]) / 2;
```

Dependences in the Iteration Space



Dependences in the Data Space



Dependences in the Matrix Addition Algorithm

```
for (unsigned int i = 0; i < N; ++i)
    for (unsigned int j = 0; j < N; ++j)
        A[i, j] = B[i, j] + C[i, j];
```

1) Is there any dependence between array accesses in matSum?

2) We can only have inter loop dependences in matSum. Why?

3) Furthermore, dependences, if any, can only exist between accesses of array A. Why?

Dependences in the Matrix Addition Algorithm

```
for (unsigned int i = 0; i < N; ++i)
    for (unsigned int j = 0; j < N; ++j)
        A[i, j] = B[i, j] + C[i, j];
```

We have only one access of any array within an iteration, so, if we have any dependence, it ought to be inter loop.

Arrays B and C are only read, so there is no way we can have a dependence between any of these accesses. In the end, we can only have inter loop dependences on A[i, j]. Let's assume that we have two iterations, i, and i', that reuse the same position of A:

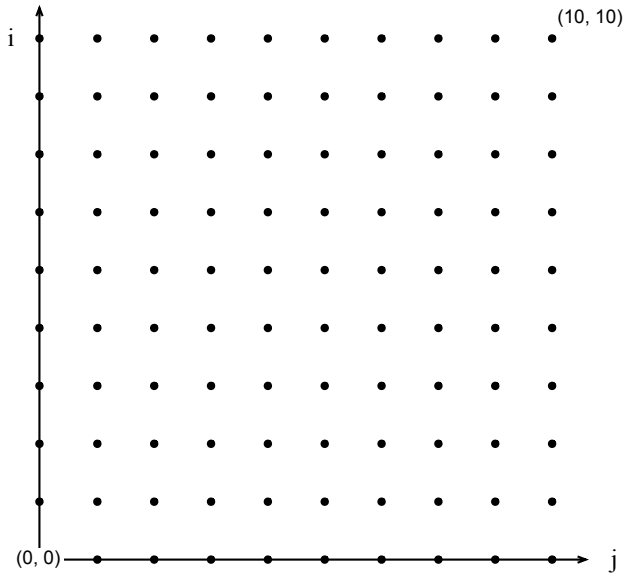
$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} i' \\ j' \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The only way we can solve this system is to have $i = i'$ and $j = j'$. In other words, the dependence would have to happen in the same iteration. But we already know that we cannot have reuse of A[., .] in the same iteration.

In the end, how is the iteration space of matSum?

Dependences in Matrix Multiplication

```
N = 10;
for (unsigned int i = 0; i < N; ++i)
   for (unsigned int j = 0; j < N; ++j)
      A[i, j] = B[i, j] + C[i, j];
```




```
for (unsigned int i = 0; i < N; ++i)
   for (unsigned int j = 0; j < N; ++j) {
      A[i, j] = .0;
      for (unsigned int k = 0; k < N; ++k)
         A[i, j] += B[i, k] * C[k, j];
   }
```

1) What about matMul, can we have dependences in this routine?

2) On which arrays?

Reuse without Dependences

```
for (unsigned int i = 0; i < N; ++i)
  for (unsigned int j = 0; j < N; ++j) {
    A[i, j] = .0;
    for (unsigned int k = 0; k < N; ++k)
      A[i, j] += B[i, k] * C[k, j];
  }
```

We cannot have dependences between accesses of arrays B and C, because these arrays are only read. Notice that we do have a lot of reuse of each position of each of these arrays. This is an example of reuse without dependences.

As we had seen before, each position of arrays B or C is reused O(N) times, because these accesses happen on a 2D data-space, within a 3D iteration space.

So, in the end, we can only have dependences between accesses of A[i, j]. We already know that we have a lot of reuse of each A[i, j]. We also know that each of these reuses is a write. Thus, we certainly have dependences…

1) Which kind of dependence do we have: true, anti or output?
2) Between which iterations do we have dependences?

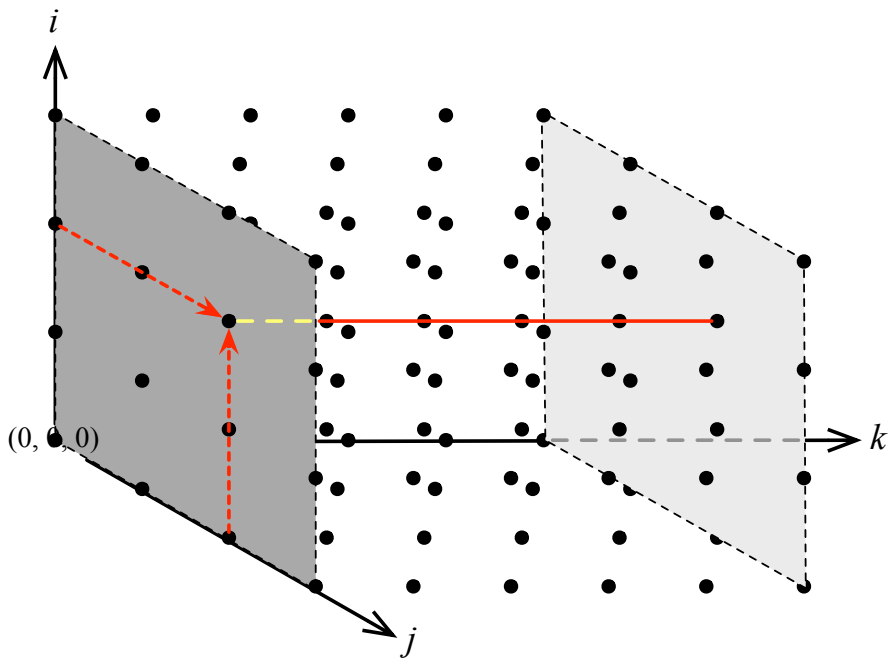
Reuse without Dependences

```
for (unsigned int i = 0; i < N; ++i)
   for (unsigned int j = 0; j < N; ++j) {
      A[i, j] = .0;
      for (unsigned int k = 0; k < N; ++k)
         A[i, j] += B[i, k] * C[k, j];
   }
```

To find all the dependences in accesses to A[i, j], we must solve a simple linear system:

$$\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix}\times\begin{bmatrix}i\\ j\\ k\end{bmatrix}+\begin{bmatrix}0\\ 0\\ 0\end{bmatrix}=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix}\times\begin{bmatrix}i'\\ j'\\ k'\end{bmatrix}+\begin{bmatrix}0\\ 0\\ 0\end{bmatrix}$$

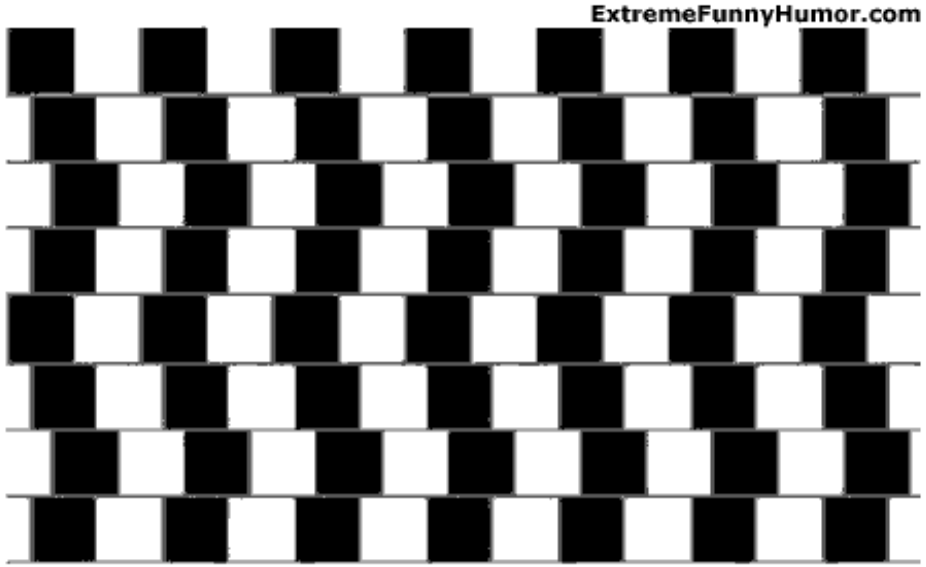
This system has solution if $i = i', j = j'$, and $0k = 0k'$. We can solve $0k = 0k'$ for any k and k'. This means that we have dependences between any two iterations on the k dimension. That is intuitively true, because if we fix i and j, by varying k we are writing on the same position of A. Thus, assigning each line on the k dimension to a different thread is a suitable parallelization of matMul.



Compilers Laboratory

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSIDADE FEDERAL DE MINAS GERAIS
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

SYNC-FREE PARALLELIZATION



Are the horizontal lines parallel or do they slope?

Assigning Computation to Threads

- We already know how to estimate reuse, and…
  - how to find dependences between different iterations of a loop.
- Now, we shall see how to assign computations, i.e., statements, including whole loops, to processes.
- If different threads do not need to communicate to solve a problem, we call this problem *synchronization free*.
- We will be producing only synchronization free code.
- We could, in principle, assign the entire algorithm that we want to parallelize to a single thread.
  - The resulting code would be trivially synchronization free.
  - But it would not be parallel…

Can you devise a way to assign computations to threads that yields synchronization free code?

Commutative Dimensions of the Iteration Space

- We say that a dimension of the iteration space is commutative if there are no dependences between points in this iteration space along that dimension.
  - Indices in that dimension can be solved in any order, so the name *commutative*.

1) How does this notion helps us to find synchronization free parallelism?

2) Can you spot the commutative dimensions in the examples on the right?

```
for (unsigned int i = 0; i < N; ++i)
   A[i] = B[i] + C[i];
```

```
for (unsigned int i = 0; i < N; ++i)
   for (unsigned int j = 0; j < N; ++j)
      A[i, j] = B[i, j] + C[i, j];
```

```
for (i = 0; i < N; i++)
   for (j = 1; j < N; j++)
      a[i, j] = (a[i, j] + a[i, j – 1]) / 2;
```

```
for (unsigned int i = 0; i < N; ++i)
   for (unsigned int j = 0; j < N; ++j) {
      A[i, j] = .0;
      for (unsigned int k = 0; k < N; ++k)
         A[i, j] += B[i, k] * C[k, j];
   }
```

Commutative Dimensions of the Iteration Space

```
for (unsigned int i = 0; i < N; ++i)
   A[i] = B[i] + C[i];
```

$$[1]\times[i]+[0]=[1]\times[i']+[0]$$

$$i = i'$$

```
for (unsigned int i = 0; i < N; ++i)
   for (unsigned int j = 0; j < N; ++j)
      A[i, j] = B[i, j] + C[i, j];
```

$$\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\times\begin{bmatrix}i\\ j\end{bmatrix}+\begin{bmatrix}0\\ 0\end{bmatrix}=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\times\begin{bmatrix}i'\\ j'\end{bmatrix}+\begin{bmatrix}0\\ 0\end{bmatrix}$$

$$i = i' \text{ and } j = j'$$

```
for (i = 0; i < N; i++)
   for (j = 1; j < N; j++)
      a[i, j] = (a[i, j] + a[i, j – 1]) / 2;
```

$$\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\times\begin{bmatrix}i\\ j\end{bmatrix}+\begin{bmatrix}0\\ -1\end{bmatrix}=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\times\begin{bmatrix}i'\\ j'\end{bmatrix}+\begin{bmatrix}0\\ 0\end{bmatrix}$$

$$i = i' \text{ and } j - 1 = j'$$

```
for (unsigned int i = 0; i < N; ++i)
   for (unsigned int j = 0; j < N; ++j) {
      A[i, j] = .0;
      for (unsigned int k = 0; k < N; ++k)
         A[i, j] += B[i, k] * C[k, j];
   }
```

$$\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix}\times\begin{bmatrix}i\\ j\\ k\end{bmatrix}+\begin{bmatrix}0\\ 0\\ 0\end{bmatrix}=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix}\times\begin{bmatrix}i'\\ j'\\ k'\end{bmatrix}+\begin{bmatrix}0\\ 0\\ 0\end{bmatrix}$$

$$i = i' \text{ and } j = j' \text{ and } 0k = 0k'$$

Commutative Dimensions of the Iteration Space

If the only possible dependence is between an iteration and itself, then that dimension is commutative. In algebraic terms, we observe this behavior when we have an equality such as $i = i'$ in the solution of the constraint system of dependences.

$$[1]\times[i]+[0]=[1]\times[i']+[0]$$

$$\boxed{i = i'}$$

$$\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\times\begin{bmatrix}i\\ j\end{bmatrix}+\begin{bmatrix}0\\ 0\end{bmatrix}=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\times\begin{bmatrix}i'\\ j'\end{bmatrix}+\begin{bmatrix}0\\ 0\end{bmatrix}$$

$$\boxed{i = i'} \text{ and } \boxed{j = j'}$$

$$\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\times\begin{bmatrix}i\\ j\end{bmatrix}+\begin{bmatrix}0\\ -1\end{bmatrix}=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\times\begin{bmatrix}i'\\ j'\end{bmatrix}+\begin{bmatrix}0\\ 0\end{bmatrix}$$

$$\boxed{i = i'} \text{ and } j - 1 = j'$$

$$\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix}\times\begin{bmatrix}i\\ j\\ k\end{bmatrix}+\begin{bmatrix}0\\ 0\\ 0\end{bmatrix}=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix}\times\begin{bmatrix}i'\\ j'\\ k'\end{bmatrix}+\begin{bmatrix}0\\ 0\\ 0\end{bmatrix}$$

$$\boxed{i = i'} \text{ and } \boxed{j = j'} \text{ and } 0k = 0k'$$

Ok, we already know how to find out commutative dimensions. So, how to use this information to assign computations to threads? In other words, what is the relation between thread identifiers and iteration points?

Mapping Iterations to Thread Identifiers

```
for (unsigned int i = 0; i < N; ++i)
   A[i] = B[i] + C[i];
```

$i = i'$

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
A[i] = B[i] + C[i];
```

```
for (unsigned int i = 0; i < N; ++i)
   for (unsigned int j = 0; j < N; ++j)
      A[i, j] = B[i, j] + C[i, j];
```

$i = i'$ and $j = j'$

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
A[i, j] = B[i, j] + C[i, j];
```

```
for (i = 0; i < N; i++)
   for (j = 1; j < N; j++)
      a[i, j] = (a[i, j] + a[i, j – 1]) / 2;
```

$i = i'$ and $j - 1 = j'$

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
for (int j = 0; j < N; ++j) {
  a[i, j] = (a[i, j] + a[i, j – 1]) / 2;
}
```

```
for (unsigned int i = 0; i < N; ++i)
   for (unsigned int j = 0; j < N; ++j) {
      A[i, j] = .0;
      for (unsigned int k = 0; k < N; ++k)
         A[i, j] += B[i, k] * C[k, j];
   }
```

$i = i'$ and $j = j'$ and $0k = 0k'$

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
A[i, j] = .0;
for (int k = 0; k < N; ++k) {
  A[i, j] += B[i, k] * C[k, j];
}
```

Coarser Grain Parallelism

We have shown how to parallelize nested loops along all their commutative dimensions. However, we do not have to create threads for all the points in the iteration space. In usual multi-core architectures, it is often the case that we do not have that many processors anyway. So, we can be a bit coarser, parallelizing only along a few commutative dimensions.

Matrix Multiplication Routine

```
for (unsigned int i = 0; i < N; ++i)
   for (unsigned int j = 0; j < N; ++j) {
      A[i, j] = .0;
      for (unsigned int k = 0; k < N; ++k)
         A[i, j] += B[i, k] * C[k, j];
   }
```

2D Parallelization of Matrix Multiplication

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
A[i, j] = .0;
for (int k = 0; k < N; ++k) {
  A[i, j] += B[i, k] * C[k, j];
}
```

For instance, matrix multiplication has two commutative dimensions. This code parallelizes each of these dimensions. But we could have chosen to parallelize only one dimension.

How could we parallelize matrix multiplication along only one of its two commutative dimensions?

Coarser Grain Parallelism

2D Parallelization of Matrix Multiplication

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
A[i, j] = .0;
for (int k = 0; k < N; ++k) {
  A[i, j] += B[i, k] * C[k, j];
}
```

1D Parallelization of Matrix Multiplication

```
int j = blockIdx.x * blockDim.x + threadIdx.x;
for (int i = 0; i < N; ++i) {
  float tmp = .0;
  for (int k = 0; k < N; ++k) {
    tmp += B[i, k] * C[k, j];
  }
  A[i, j] = tmp;
}
```

In this example we chose to parallelize the j loop. In principle we could parallelize along either dimension of the iteration space. However, due to a technicality of GPUs, known as *coalesced memory access*♤, the approach that we chose is slightly better.

But, after all, which version of matrix multiplication do you think is the best one?

♤: See the NVIDIA Best Practice Guide 2.3, page 20 for more information about coalesced memory access.

Coarser Grain Parallelism

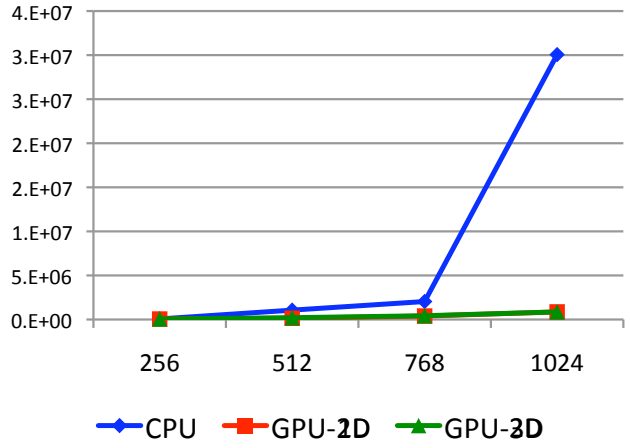
2D Parallelization of Matrix Multiplication

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
A[i, j] = .0;
for (int k = 0; k < N; ++k) {
  A[i, j] += B[i, k] * C[k, j];
}
```

1D Parallelization of Matrix Multiplication

```
int j = blockIdx.x * blockDim.x + threadIdx.x;
for (int i = 0; i < N; ++i) {
  float tmp = .0;
  for (int k = 0; k < N; ++k) {
    tmp += B[i, k] * C[k, j];
  }
  A[i, j] = tmp;
}
```

In practice, there is not much difference between these versions. However, the 1D algorithm is more likely to underuse the hardware if the block size is not a multiple of the number of physical threads available.



Tick	CPU	GPU-2D	GPU-1D
256	64213	53323	62891
512	1065125	145050	204869
768	2041679	390712	427851
1024	30059627	860797	862093

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

A Bit of History

- A lot of research has been done on automatic parallelization of loops, mostly in the 80's and 90's.
- Lamport seems to have been the first to model loops as iterations spaces to find parallelism for multiprocessors.
- Many ideas used in this presentation have been taken from the PFC parallelizer, from Rice University.

- Allen, R. and K. Kennedy, "Automatic Translation of Fortran Programs to Vector form", TOPLAS 9:4 (1987), pp. 491-542
- Lamport, L., "The Parallel Execution of DO Loops", Comm. ACM 17:2 (1974), pp. 83-93
- Maydan, D. E., J. L. Hennessy, and M. S. Lam, "An Efficient Method for Exact Dependence Analysis", PLDI, (1991), pp. 1-14
- Allen, F. E., M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An Overview of the PTRAN Analysis System for Multiprocessing", J. Parallel and Distributed Computing 5:5 (1988), pp. 617-640