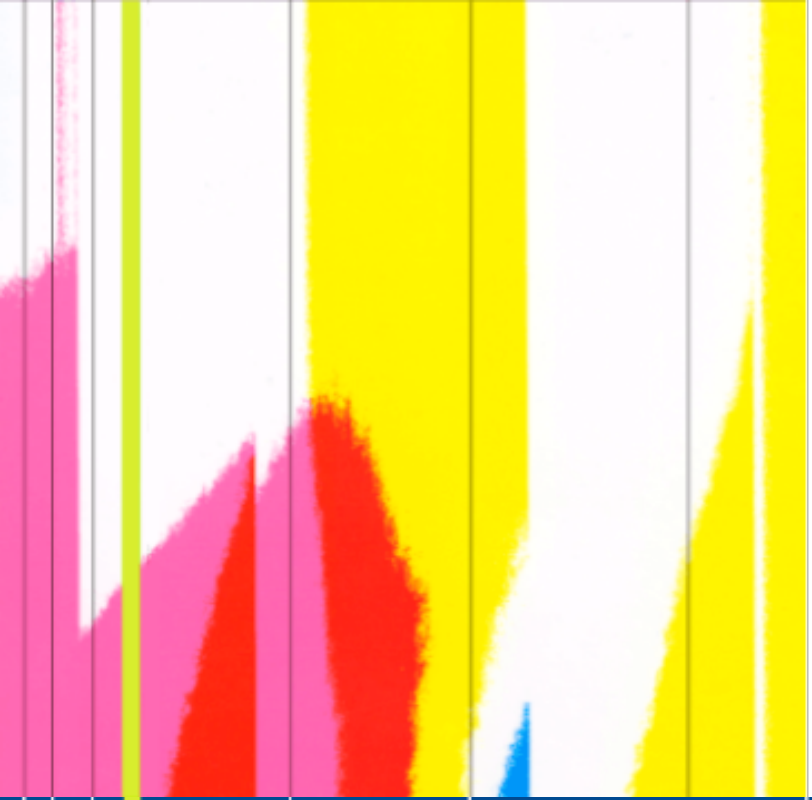


GPU for Dummies

Cédric Castagnède - François Rué
7 juin 2011

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
BORDEAUX - SUD-OUEST

Outline

- CPU versus GPU: deep differences
  - CPU Architecture
  - GPU Architecture (NVIDIA Card)
  - We need new programming language... again...

- CUDA: an API with new concepts
  - Grid and blocs
  - Memory space overview
  - The useful functions
  - Extensions and Kernels
  - Difference between CUDA C and Fortran

Outline

- Why make it simple when it can be complicated?
  - Example 1: Transpose matrix
  - Several implementations
  - Example 2:Matrix multiplication
  - Several implementations

- Help me! Where are the libraries?
  - Libraries for CUDA?
  - Do not you see BLAS right there?
  - CUBLAS help to find performance
  - Outcome of all these sheets

Outline

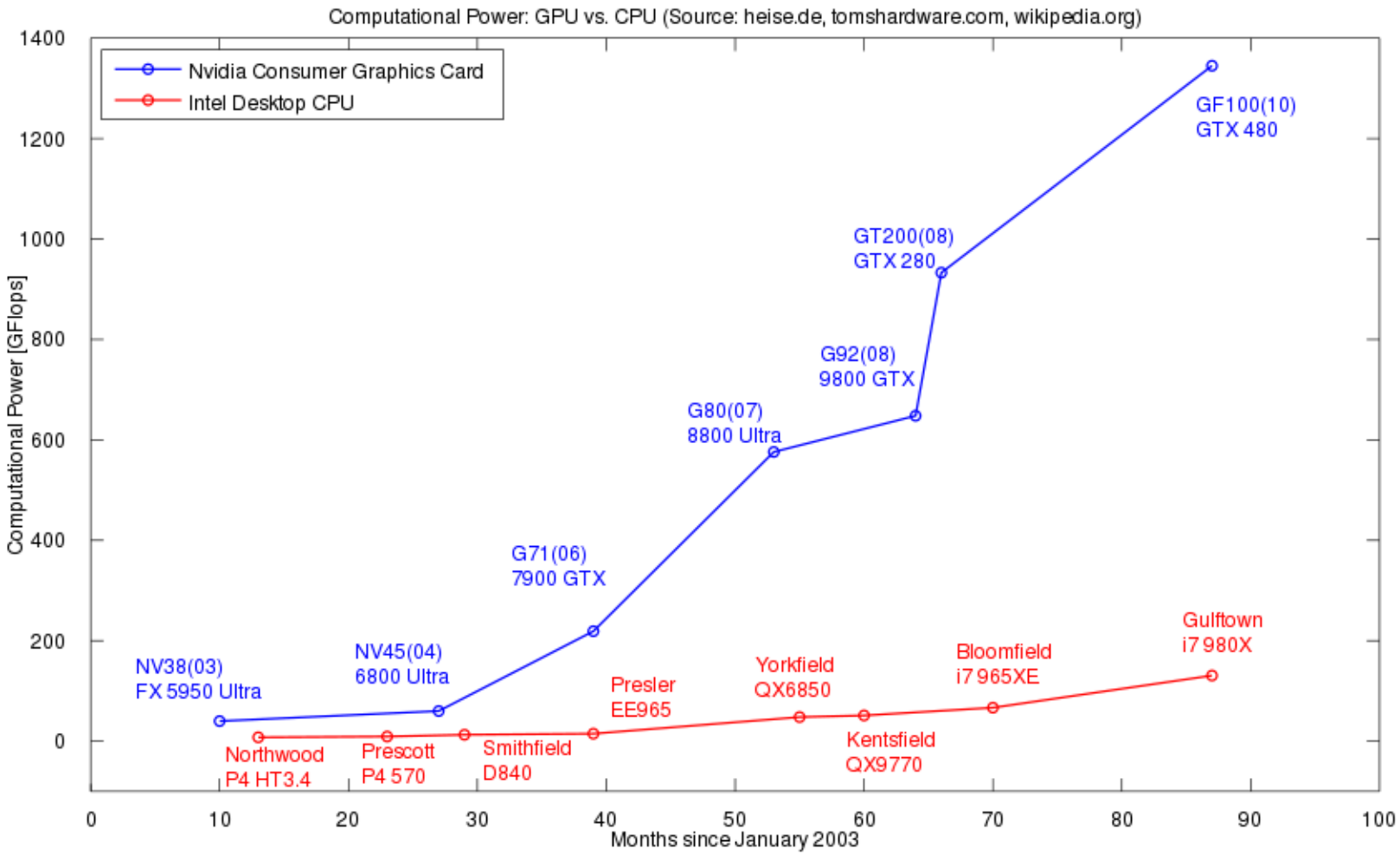
- And in real life...
  - You have a CPU implementation
  - Find out where you can place libraries calls
  - What we need to run on GPU
  - Outline

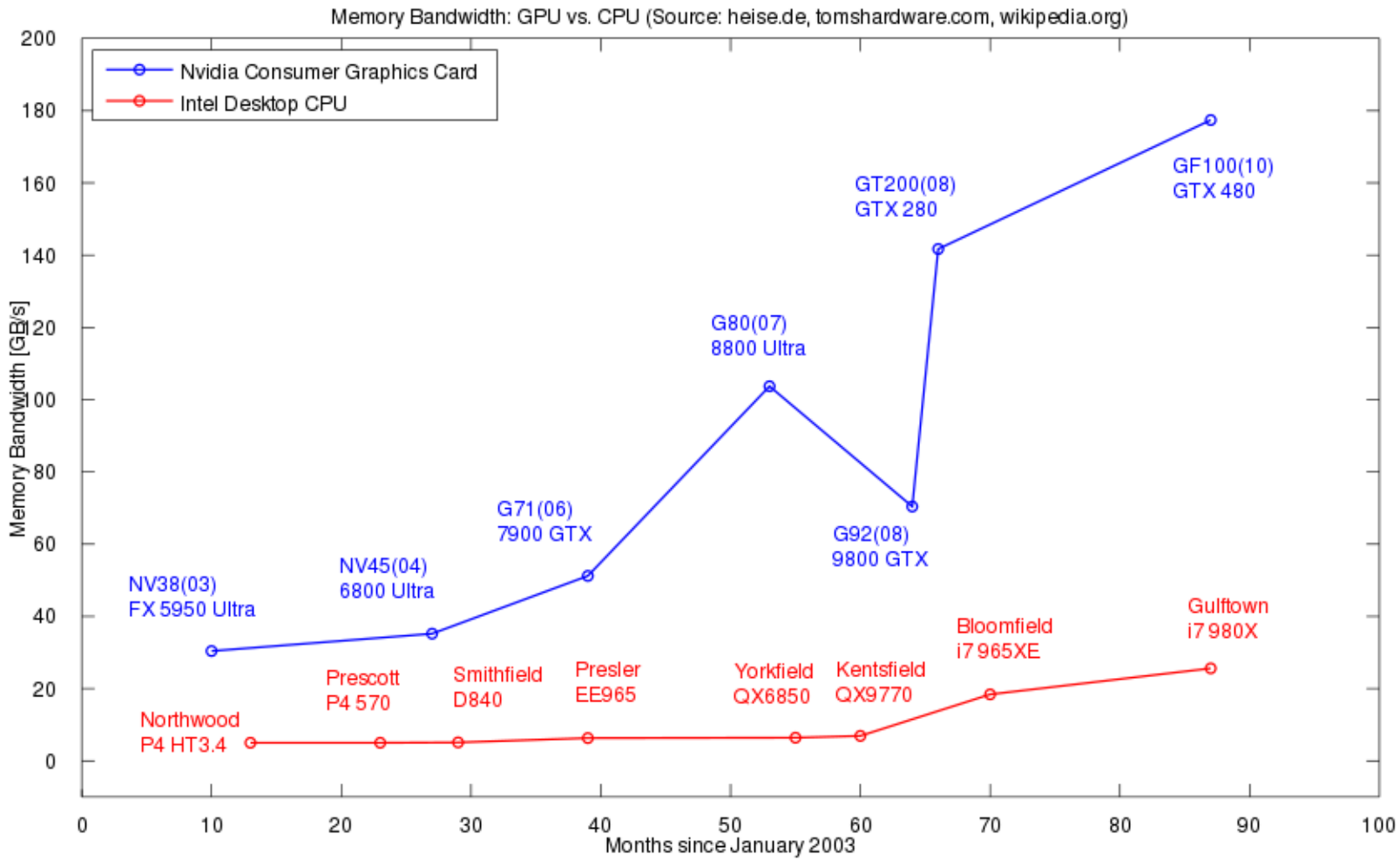
- Outlines
  - When going on GPU is profitable?
  - Go on GPU at your speed
  - Your implementation will be portable?
  - What is your future?

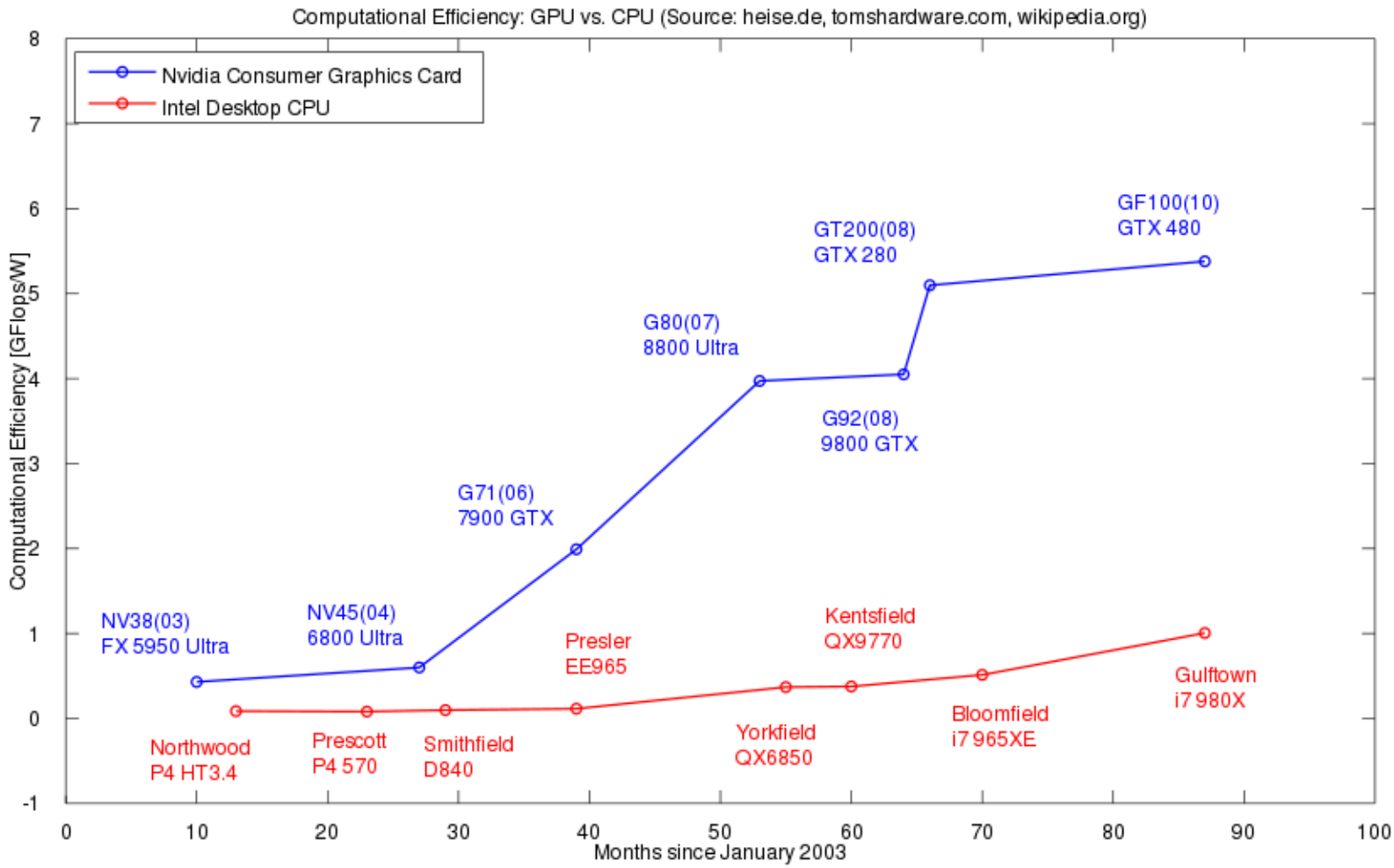
Why use GPU?

Because...

- GPU is very popular
  - thanks to video game
  - attractive market
- GPU are massively parallel
  - large local memory
  - high bandwidth memory
  - lot of cores and ALU
- Bandwidth is very important
- GPU cards are massively produced, so cluster cost is reduced
- Energy Efficiency is over 6 Gflops / watts


Computational Power: GPU vs. CPU (Source: heise.de, tomshardware.com, wikipedia.org)
Nvidia Consumer Graphics Card
Intel Desktop CPU
Computational Power [GFlops]
Months since January 2003
0
200
400
600
800
1000
1200
1400
0
10
20
30
40
50
60
70
80
90
100
NV38(03)
FX 5950 Ultra
NV45(04)
6800 Ultra
G71(06)
7900 GTX
G80(07)
8800 Ultra
G92(08)
9800 GTX
GT200(08)
GTX 280
GF100(10)
GTX 480
Northwood
P4 HT3.4
Prescott
P4 570
Smithfield
D840
Presler
EE965
Yorkfield
QX6850
Kentsfield
QX9770
Bloomfield
i7 965XE
Gulftown
i7 980X

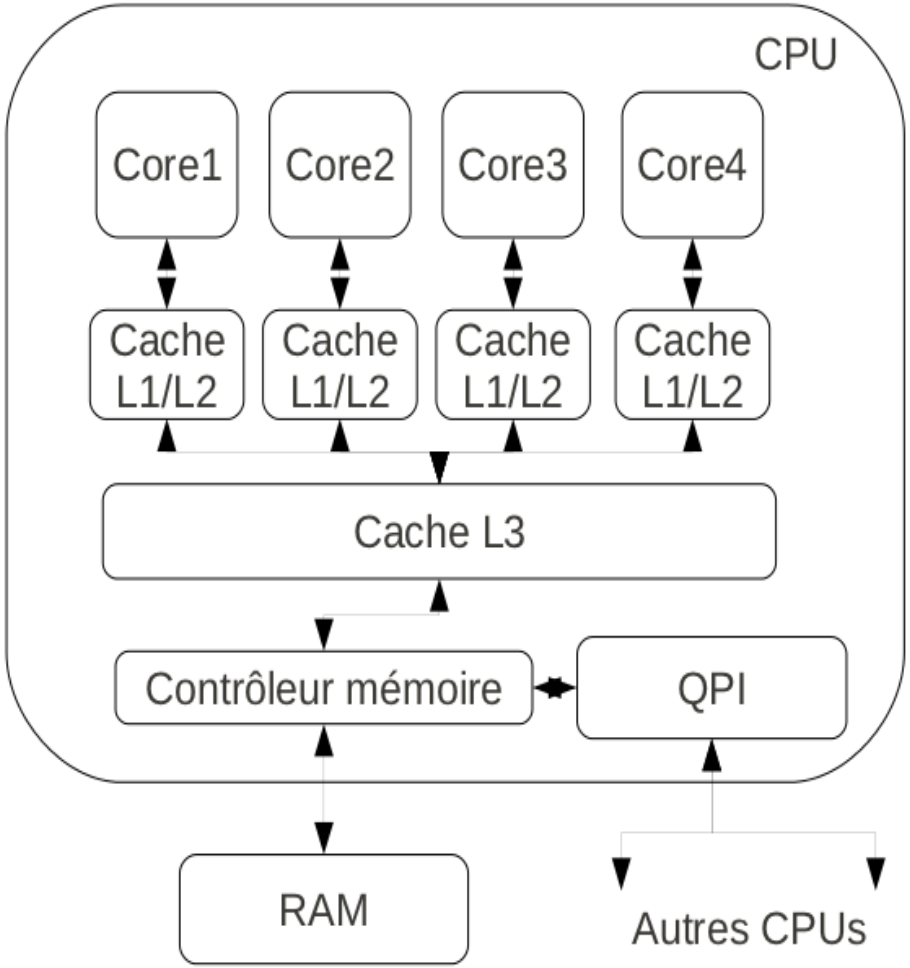

Memory Bandwidth: GPU vs. CPU (Source: heise.de, tomshardware.com, wikipedia.org)
Nvidia Consumer Graphics Card
Intel Desktop CPU
Memory Bandwidth [GB/s]
Months since January 2003
0
20
40
60
80
100
120
140
160
180
200
0
10
20
30
40
50
60
70
80
90
100
NV38(03)
FX 5950 Ultra
NV45(04)
6800 Ultra
G71(06)
7900 GTX
G80(07)
8800 Ultra
G92(08)
9800 GTX
GT200(08)
GTX 280
GF100(10)
GTX 480
Northwood
P4 HT3.4
Prescott
P4 570
Smithfield
D840
Presler
EE965
Yorkfield
QX6850
Kentsfield
QX9770
Bloomfield
i7 965XE
Gulftown
i7 980X


Computational Efficiency: GPU vs. CPU (Source: heise.de, tomshardware.com, wikipedia.org)
Nvidia Consumer Graphics Card
Intel Desktop CPU
Computational Efficiency [GFlops/W]
Months since January 2003
NV38(03)
FX 5950 Ultra
NV45(04)
6800 Ultra
G71(06)
7900 GTX
G80(07)
8800 Ultra
G92(08)
9800 GTX
GT200(08)
GTX 280
GF100(10)
GTX 480
Northwood
P4 HT3.4
Prescott
P4 570
Smithfield
D840
Presler
EE965
Yorkfield
QX6850
Kentsfield
QX9770
Bloomfield
i7 965XE
Gulftown
i7 980X

CPU versus GPU: deep differences

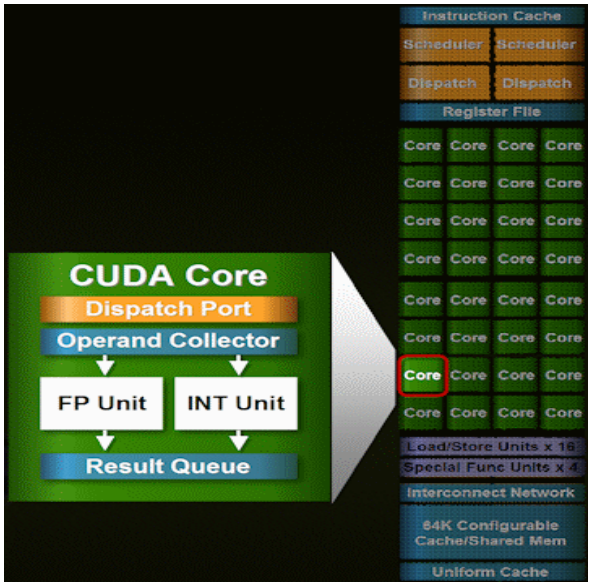
CPU Architecture

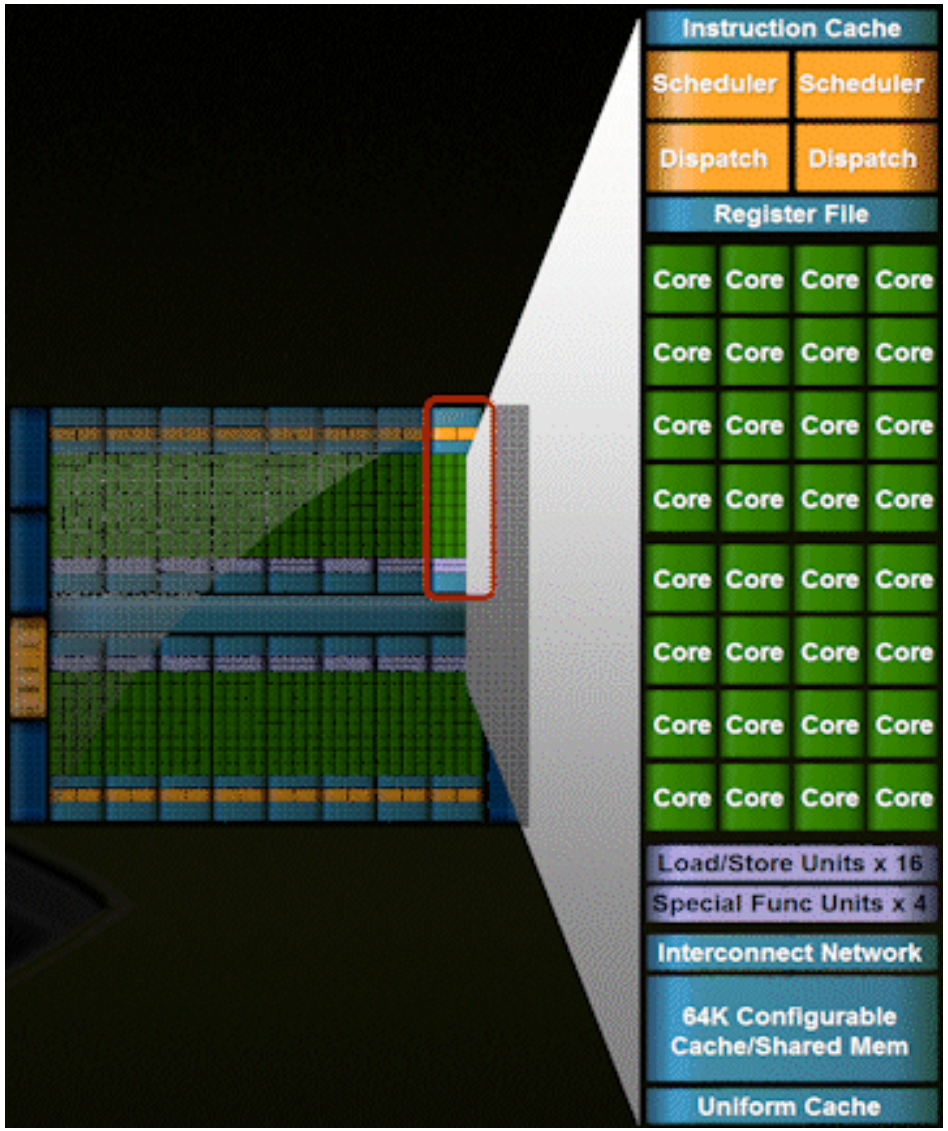
- core i7
  - Real quadri cores
  - Hyper Threading
    - 8 threads simultaneously
  - QPI
    - Point to point interconnection
    - Up to 4 CPU
    - 4 * 4 * 2 = 32 threads simultaneously



GPU Architecture (NVIDIA Card)

- SP (1 scalar)
- Streaming Multiprocessor SM
  - 32 SP
  - 4SFU
  - M-Thread Ctl
  - Shared Memory





We need new programming language... again...

- To exploit all resources of these hardware and facilitate programming
- Several solutions are available:
  - NVIDIA CUDA C
    - provides a low level and a high level API
    - works only on NVIDIA Card
  - OpenCL
    - frameworks for writing programs
    - can be executed across heterogeneous platforms consisting of CPUs, GPUs, and other processors
  - PGI CUDA Fortran
    - wrappers of NVIDIA CUDA C
    - tool chain for programming in Fortran
  - PGI Accelerator
    - implementation by directives (like OpenMP)
    - easy-to-begin

CUDA: an API with new concepts

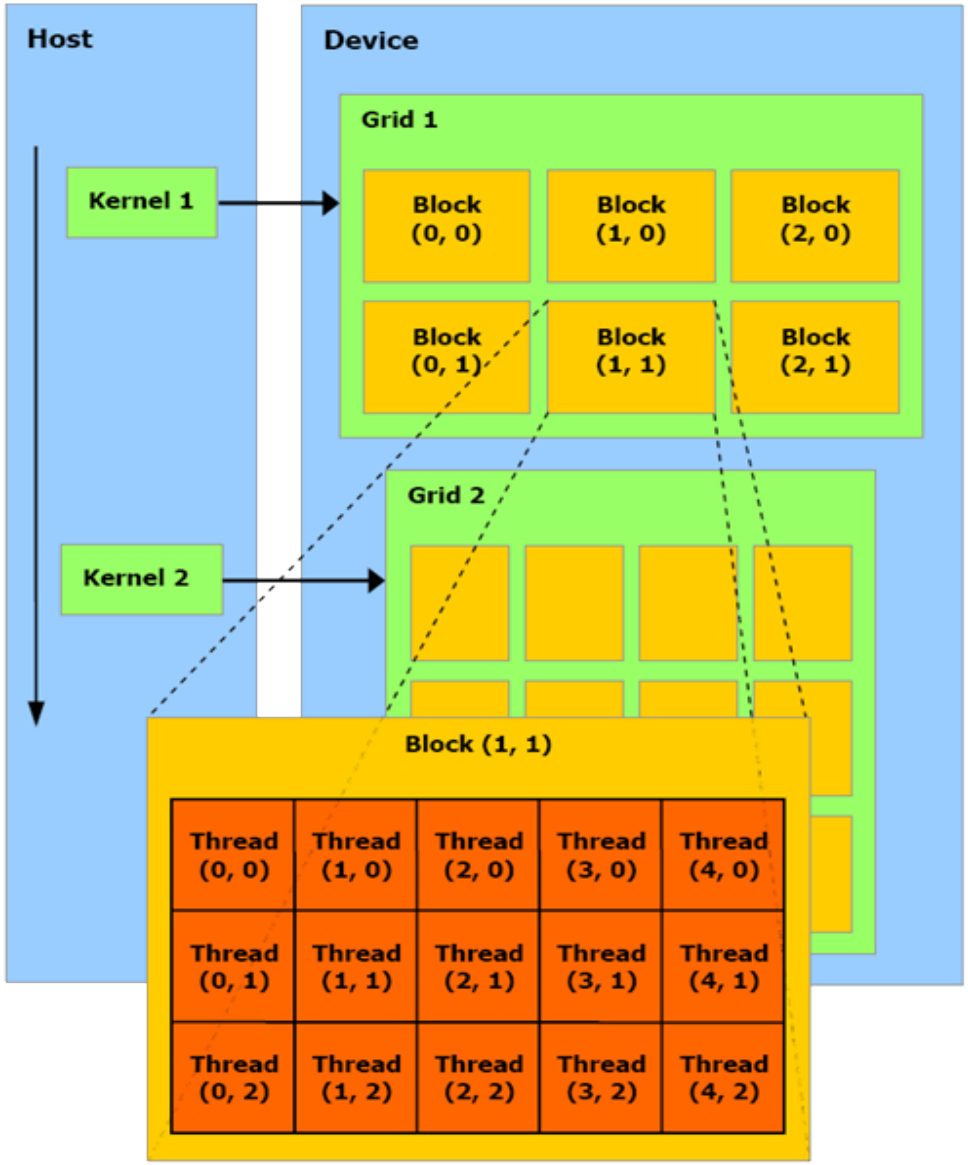
INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
BORDEAUX - SUD-OUEST

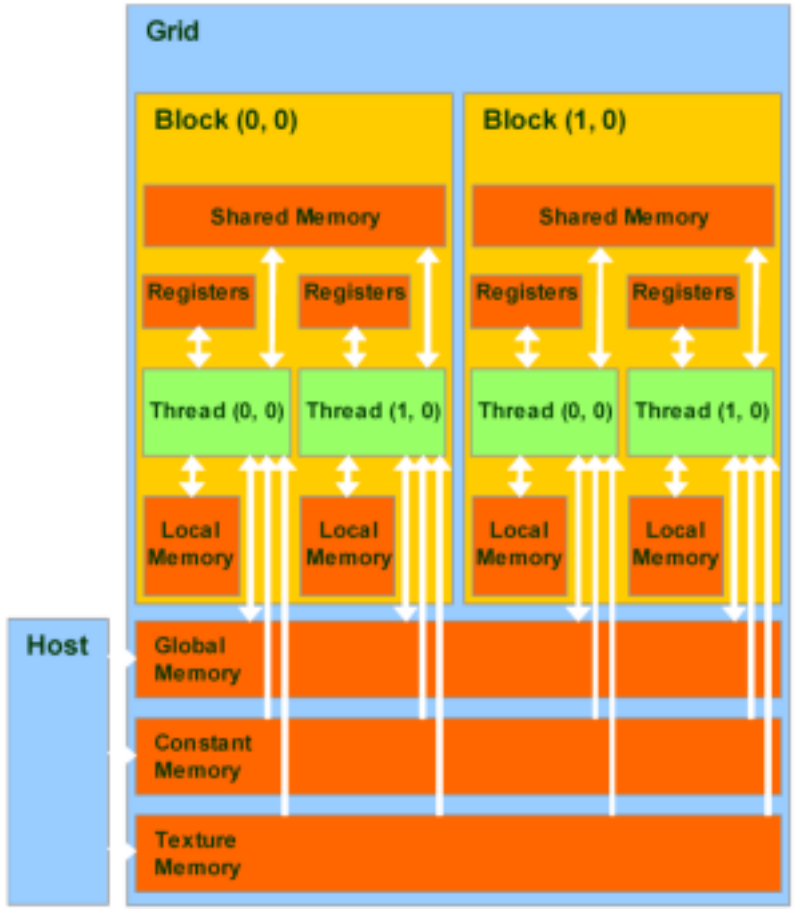
Grid and blocs

- Grid
  - set of blocks
- Blocks
  - set of threads
  - each block are independent
  - each blocks have a specific shared memory
- Threads
  - each thread runs an instance of kernel
  - threads can be synchronized in a block
  - all thread can exchange data thanks to shared memory
- Warp
  - set of n threads (n=32 for C2070)
  - these n threads are run simultaneously
  - a warp is run over 2 cycles



Memory space overview

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can R/W global, constant, and texture memories



The useful functions

CUDA C:

- Memory management:
  - cudaMalloc(&Md, size * sizeof * Md);
  - cudaMemcpy(Md, M, size * sizeof * Md);
  - cudaFree(Md);
- Device management:
  - cudaSetDevice (devnum);
  - cudaGetDevice (devnum);
  - cudaGetDeviceProperties (prop, devnum);

CUDA Fortran:

- Memory management:
  - call cudaMalloc(devptr, count)
  - call cudaMemcpy(dst, src, count, kdir)
  - call cudaFree(devptr)
- Device management:
  - call cudaSetDevice(devnum)
  - call cudaGetDevice(devnum)
  - call cudaGetDeviceProperties (prop, devnum)

Extensions and Kernels (1/2)

CUDA C:

- Function qualifiers:
  - __global__ void kernel() { }
  - __device__ void fromdev() {}
  - __host__ void fromhost(){}
- Variable qualifiers:
  - __shared__ float tile[][];
  - __device__ int GlobalVar;
  - __constant__ int Var;

CUDA Fortran:

- Subroutine qualifiers:
  - attributes(global) subroutine my_kernel (...)
  - attributes(device) subroutine my_kernel (...)
  - attributes(host) subroutine my_kernel (...)
- Variable qualifiers:
  - <type>, device :: x
  - <type>, constant :: x
  - <type>, shared :: x
  - <type>, pinned :: x
  - <type>, value :: x

Extensions and Kernels (2/2)

CUDA C:

- Built-in variables and functions valid in device code:
  - dim3 bDim, gDim;
  - gDim(32,32,0);
  - bDim(32,32,32);
  - kernel<<<gDim, bDim>>>(...);
- Execution configurations:
  - gridDim.{x, y, z}
  - blockDim.{x,y,z}
  - blockIdx.{x,y,z}
  - threadIdx.{x, y, z}
  - __syncthreads();

CUDA Fortran:

- Built-in variables and functions valid in device code:
  - type(dim3) :: dimGrid, dimBlock
  - dimGrid = dim3(32, 32, 0)
  - dimBlock = dim3(32, 32, 32)
  - call my_kernel <<<dimGrid, dimBlock>>> (...)
- Execution configurations:
  - gridDim%{x, y, z}
  - blockDim%{x, y, z}
  - blockIdx%{x, y, z}
  - threadIdx%{x, y, z}
  - call syncthreads()

Compilation

CUDA C:

- Compiler + CUDA Toolkit
- file: .cu
  - compiler: nvcc
  - link: cudart
- nvcc *.cu -lcudart -o prog

PGI CUDA Fortran:

- Only PGI Compiler
- file: .cuf .CUF
  - compiler:pgfortran
  - link: cuda
- pgfortran -Mcuda *.cuf -o prog

Differences between CUDA C and Fortran

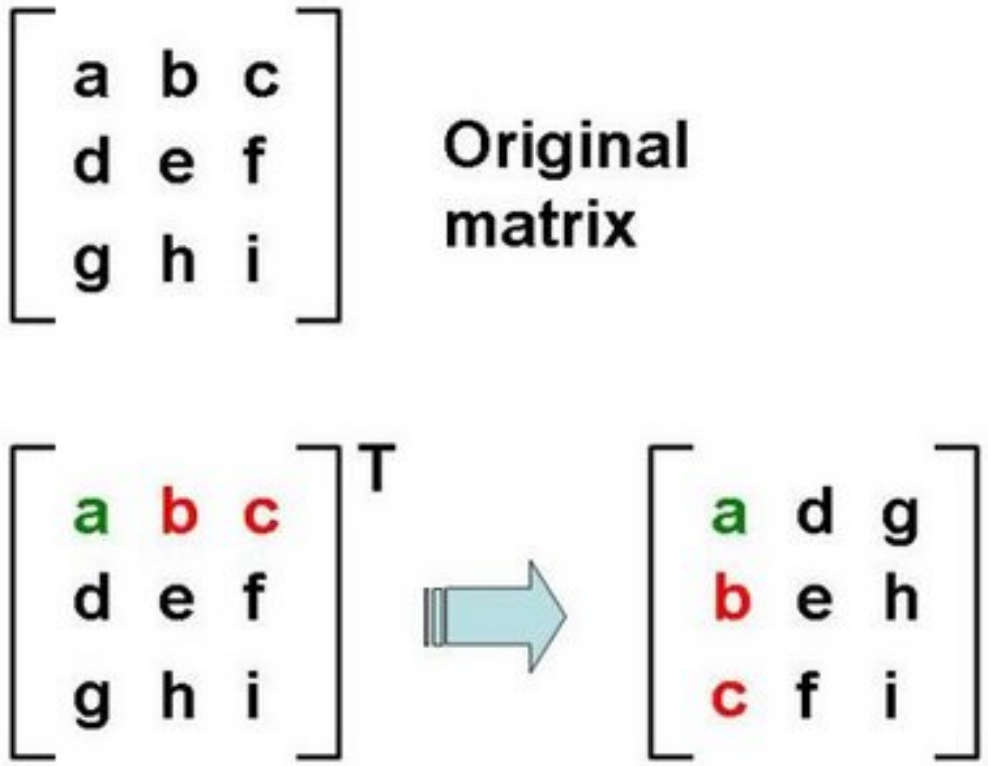
- CUDA C supports
  - texture memory
  - Runtime API
  - Drivers API
- Interoperability with
  - OpenGL
  - Direct3D
- Indexing
  - arrays 0-based
  - threadIdx and blockIdx 0-based

- CUDA Fortran don't support
  - texture memory
  - Runtime API
  - Drivers API
- No interoperability with
  - OpenGL
  - Direct3D
- Indexing
  - arrays 1-based
  - threadIdx and blockIdx 1-based

Why make it simple when it can be complicated?

Example 1: Transpose matrix

- Example of data shaking-up
  - common in algorithms for optimizations
  - data moving only, no data treatments
- Example similar to copy
  - give a reference
  - simple to implement
- We need an indicator : bandwidth
  - here, bandwidth = [memory size of the matrix] / [time]
  - warning: effective bandwidth = [2 * memory size of the matrix] / [time]



Transpose matrix: Host code (1/2)

Source Code in CUDA C:

```
int main (int args, char* argv[])
{

const int HEIGHT = 1024;
const int WIDTH   = 1024;

const int SIZE=WIDTH*HEIGHT*sizeof(float);

float* M   = (float*)malloc(SIZE);
float* Md = NULL;
float* Id   = NULL;
cudaMalloc((void**)&Md, SIZE);
cudaMalloc((void**)&Id, SIZE);

dim3 bDim(16, 16);
dim3 gDim(WIDTH / bDim.x, HEIGHT / bDim.
y);

transpose<<<gDim, bDim>>>(Md, Bd, WIDTH);

cudaFree(Md);
cudaFree(Id);

free(M);

return 0;

}
```

Transpose matrix: Host code (2/2)

Source Code in CUDA Fortran:

```
program cuda_transpose
    use cudafor
    implicit none

    type(dim3) :: dimGrid, dimBlock
    real, allocatable, device :: an_d(:,:),at_d(:,:)
    real, allocatable :: an(:,:), at(:,:)
    real :: alpha, beta
    integer :: m, l

    m = 1024 ; l = 1024
    allocate (an(m,l), at(l,m))
    allocate (an_d(m,l), at_d(l,m))

    a = 1.0 ; b = 2.0 ; c = 3.0
    alpha = 1.0 ; beta  = 0.0

    dimGrid  = dim3(m/32, l/32, 1)
    dimBlock = dim3(32, 8, 1)

    an_d = an
    call matrix_transpose_direct<<< dimGrid,
dimBlock >>>(an_d, at_d, m, l)
    at = at_d

    deallocate (an,at)
    deallocate (an_d,at_d)
end program cuda_transpose
```

Simple-minded implementation (1/2)

Idea:

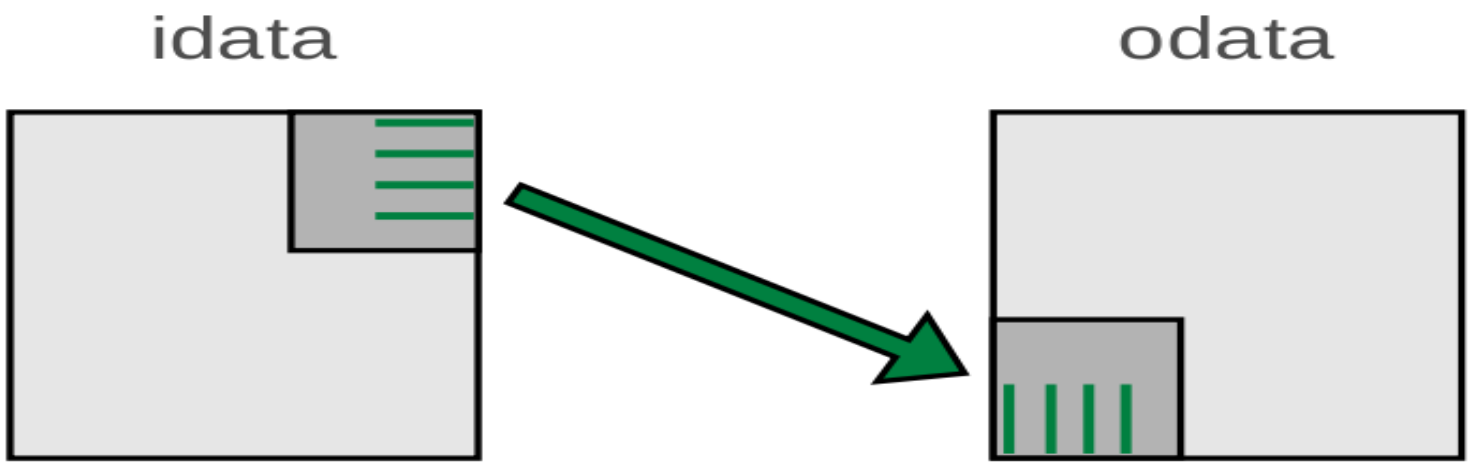
- compute four values by thread
- direct transfers from/to global memory
- kernel very similar to CPU code

Why?

- easy-to-write
- very simple (maybe too simple?)

Parameter of the kernel:

- dimGrid = (m/32, l/32, 1)
- dimBlock = (32, 8, 1)



Simple-minded implementation (2/2)

Source Code in CUDA C:

```
 __global__ void transpose
(float* in, float* out, uint width)

{

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

      out[tx * width + ty] = in[ty * width + tx];

}
```

Source Code in CUDA Fortran:

```
attributes(global) subroutine matrix_transpose
(an, at, m, n)
   implicit none
   real, intent(in) :: an(m,n)
   real, intent(out) :: at(m,n)
   integer, value :: m, n
   integer :: i, ix, iy

   ix = (blockIdx%x-1) * blockDim%x +
threadIdx%x
   iy = (blockIdx%y-1) * blockDim%y +
threadIdx%y

   do i = 0, blockDim%x-1, 8
      at(iy+i,ix) = an(ix,iy+i)
   enddo

end subroutine matrix_transpose
```

Use shared memory, it exists for a reason... (1/2)

Idea:

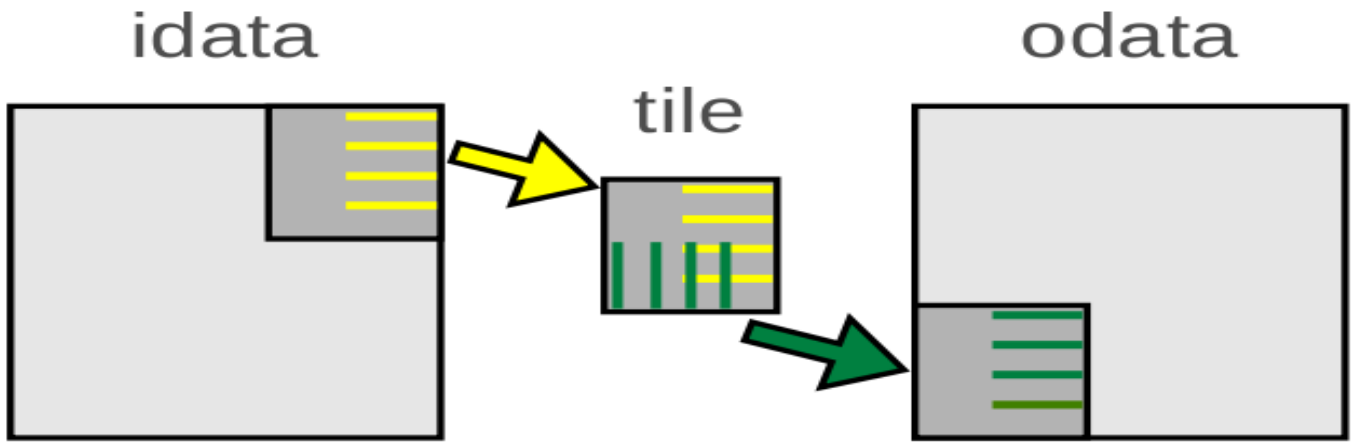
- Compute four values of C by thread again
- each thread wrtie in shared memory
- synchronize all threads
- each thread read in shared memory

Why?

- shared memory have lower latency
- shared memory have higher bandwidth
- more control on access memory

Parameter of the kernel:

- dimGrid = (m/32, l/32, 1)
- dimBlock = (32, 8, 1)



Use shared memory, it exists for a reason... (2/2)

Source Code in CUDA C:

```
 __global__
void transpose
(float* in, float* out, uint width)
{
__shared__ float tile[32][32];

__shared__ int i;
__shared__ int block;
uint tx = blockIdx.x * blockDim.x + threadIdx.x;
uint ty = blockIdx.y * blockDim.y + threadIdx.y;

block = width / 32;

for(i = 0; i < 32; i+= block)
tile[threadIdx.y + i][threadIdx.x] = in[ty * width + tx];

for(i = 0; i < 32; i+= block)
out[tx * width + ty + i * width] = tile[threadIdx.x]
[threadIdx.y + i];

}
```

Source Code in CUDA Fortran:

```
attributes(global) subroutine matrix_transpose(an, at, m, n)
  implicit none
  real, intent(in) :: an(m,n)
  real, intent(out) :: at(m,n)
  real, shared :: tile(32,32)
  integer, value :: m, n
  integer :: i, ix, iy, tx, ty

  tx  = threadIdx%x
  ty  = threadIdx%y
  ix = (blockIdx%x-1) * blockDim%x + tx
  iy = (blockIdx%y-1) * blockDim%x + ty
  do i = 0, blockDim%x, 8
       tile(tx,ty+i) = an(ix,iy+i)
  enddo
  call syncthreads()

  ix = (blockIdx%y-1) * blockDim%x + tx
  iy = (blockIdx%x-1) * blockDim%x + ty
  do i = 0, blockDim%x-1, 8
    at(ix,iy+i) = tile(ty+i,tx)
  enddo
end subroutine matrix_transpose
```

We can do more complicated... Partition camping (1/2)

Idea:

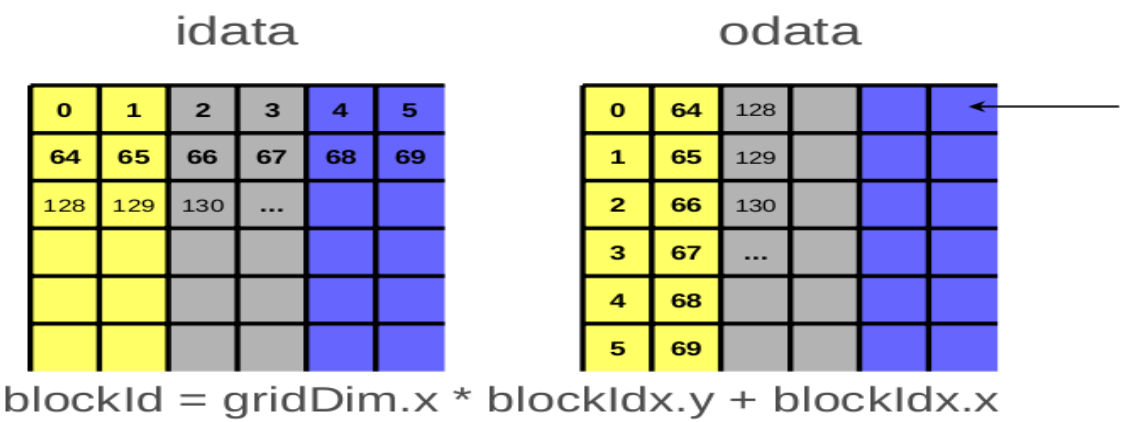
- Compute four values of C by thread again
- reordering blockIdx to force diagonalized numbering
- used of shared memroy

Why?

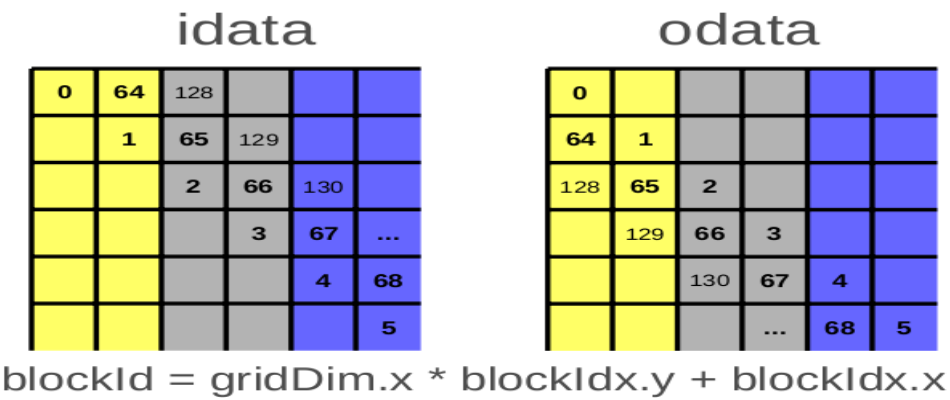
- avoid partition camping ie avoid concurrency access memory
- more control on access memory

Parameter of the kernel:

- dimGrid = (m/32, l/32, 1)
- dimBlock = (32, 8, 1)



tiles in matrices
colors = partitions

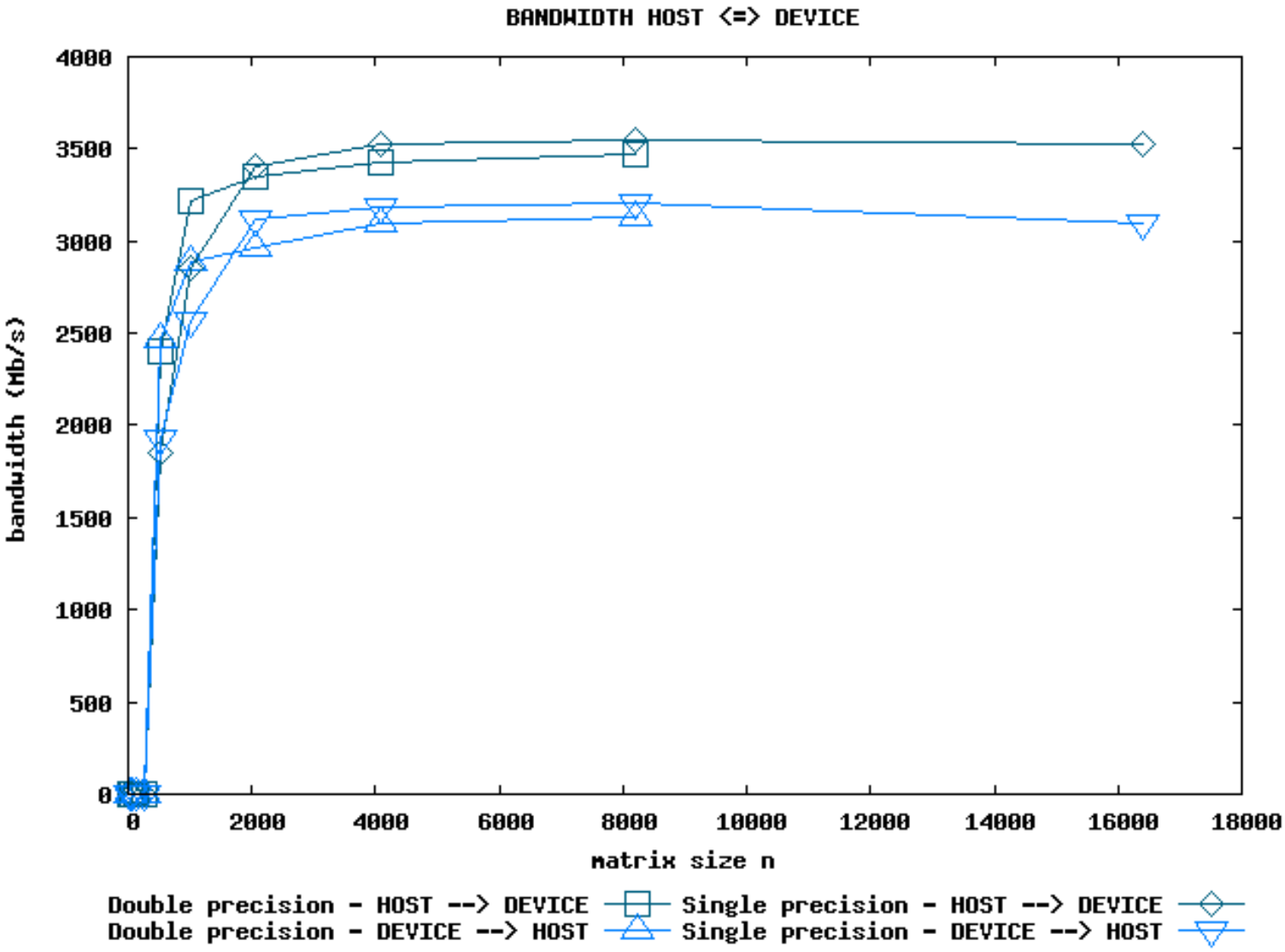


We can do more complicated... Partition camping (2/2)

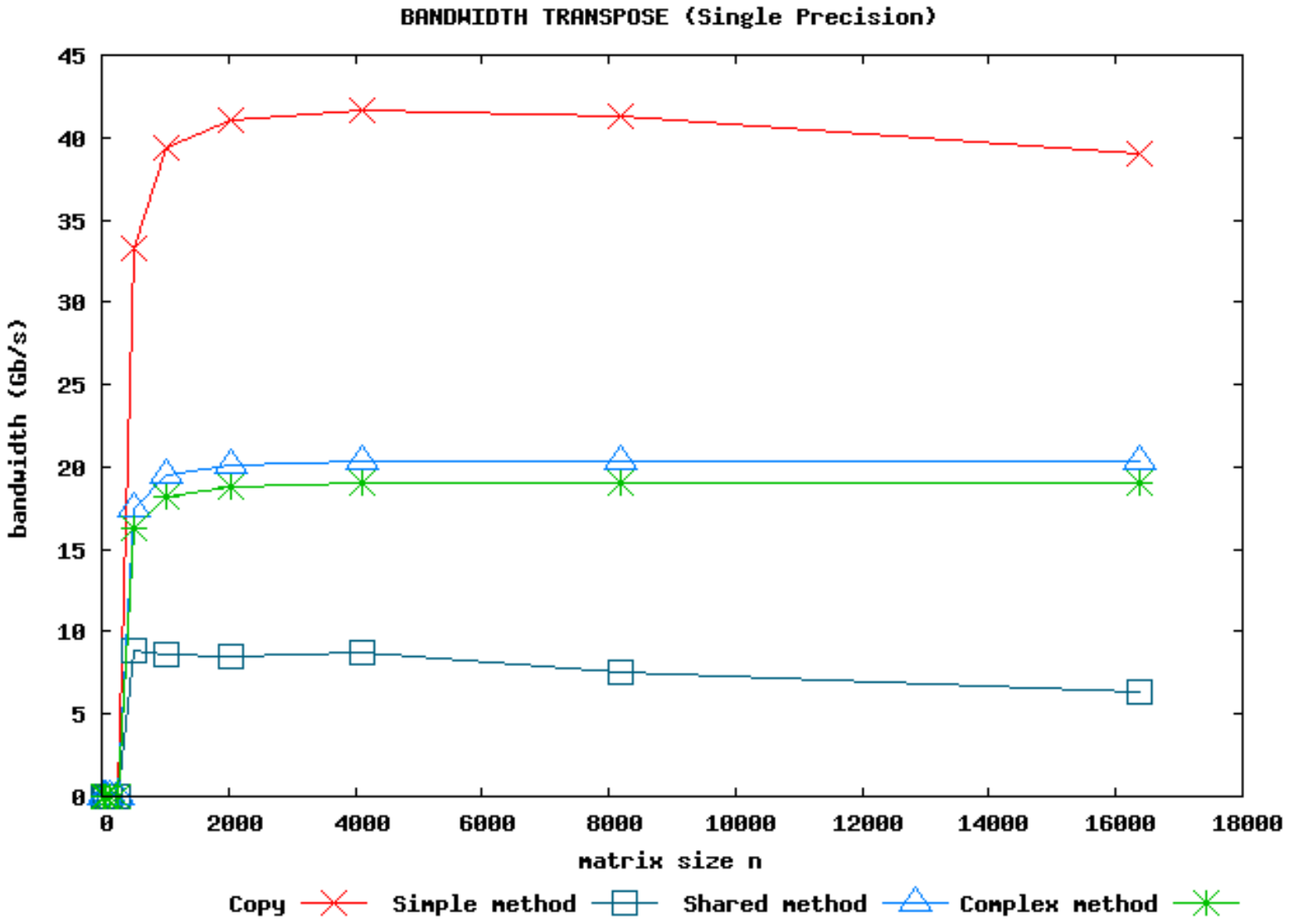
Source Code in CUDA Fortran:

```
attributes(global) subroutine matrix_transpose(an, at,
m, n)
  implicit none
  real, intent(in) :: an(m,n)
  real, intent(out) :: at(m,n)
  real, shared :: tile(32,32)
  integer, shared :: ibx, iby
  integer, value :: m, n
  integer :: i, ix, iy, tx, ty, ibid

  tx  = threadIdx%x
  ty  = threadIdx%y
  if (m==n) then
    iby = blockIdx%x-1
    ibx = mod(blockIdx%x+blockIdx%y-2, gridDim%x)
  else
    ibid = gridDim%x*(blockIdx%y-1) + blockIdx%x -
1
    iby = mod(ibid, gridDim%y)
    ibx = mod(ibid/gridDim%y+iy, gridDim%x)
  endif
  ibx = ibx * blockIdx%x
  iby = iby * blockIdx%x

  ix = ibx + tx
  iy = iby + ty
  do i = 0, blockIdx%x-1, 8
    tile(tx,ty+i) = an(ix,iy+i)
  enddo
  call syncthreads()

  ix = iby + tx
  iy = ibx + ty
  do i = 0, blockIdx%x-1,8
    at(ix,iy+i) = tile(ty+i,tx)
  enddo

end subroutine matrix_transpose
```

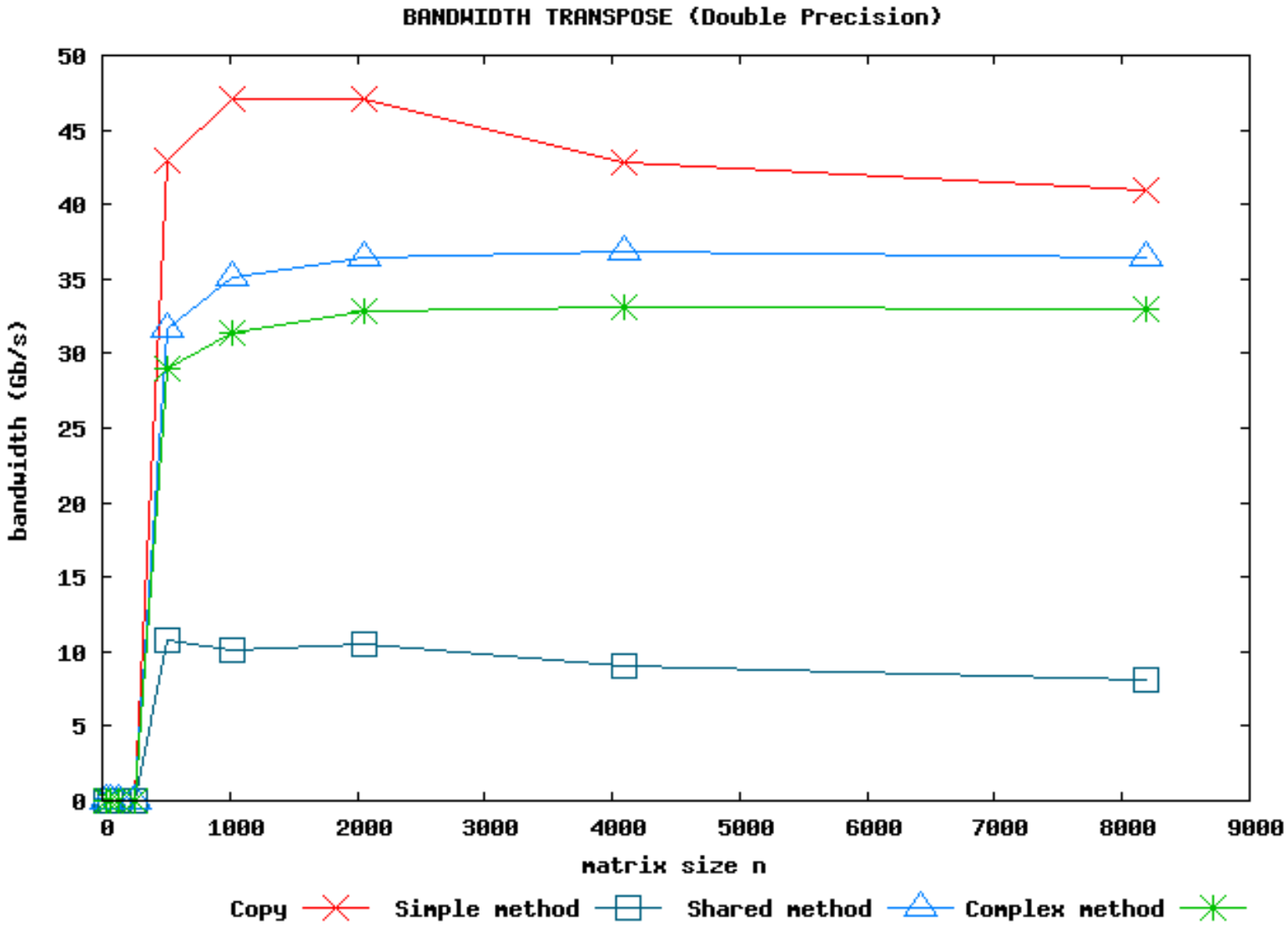
Which performance in the end?



Which performance in the end?

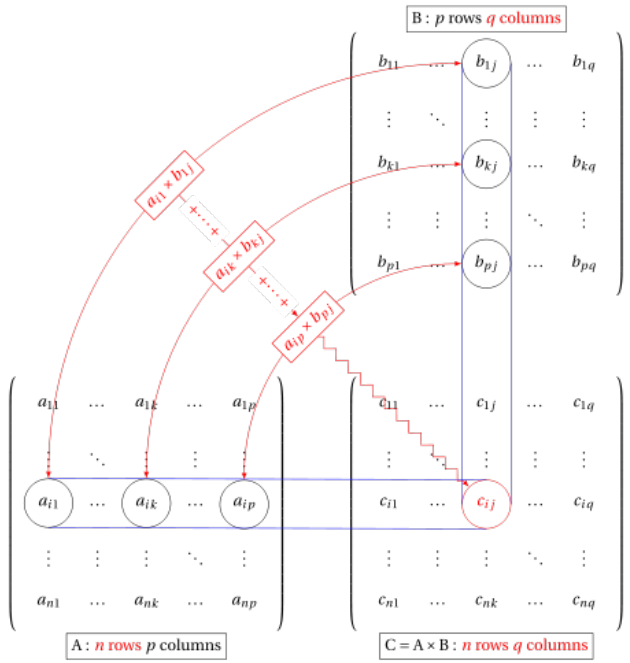


Which performance in the end?

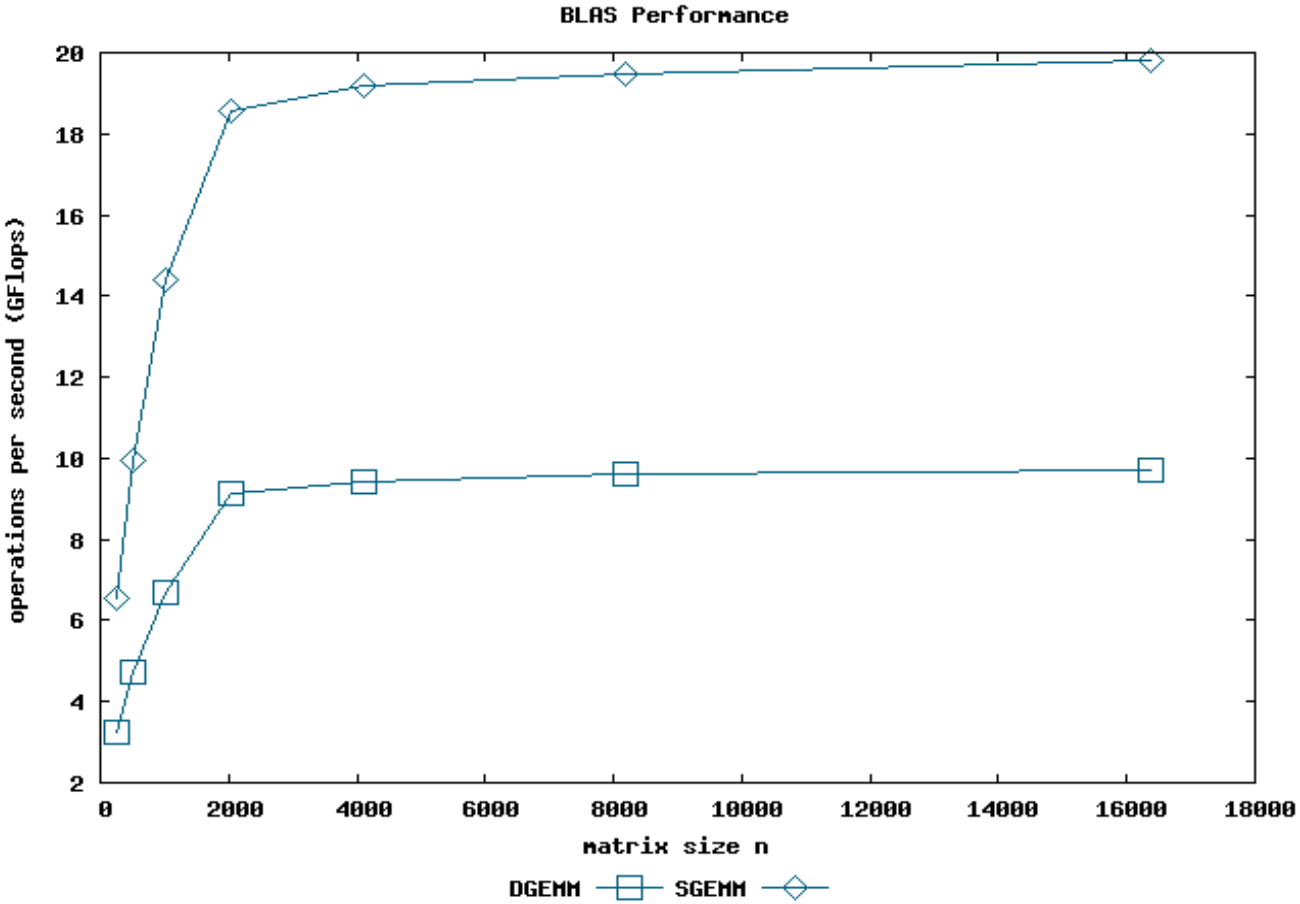


Example 2: Matrix multiplication

- Common algorithm:
  - "school example"
  - use for benchmarking
- We need an indicator : Flops or Floating Point Operations Per Second:
  - Flops = [number of operations of the algorithm] / [time]
  - here: number of operations of the algorithm= n * q * (2p+2)
  - if square matrices : complexity in $O(n^3)$



Performance on CPU



- Intel ComposerXE 12.0 + Intel MKL 10.3
- <t>GEMM in multithread version
- Test machine: one node "mirage" on PlaFRIM
- Benchmkark over 12 thread

Simple-minded implementation (1/2)

Idea:

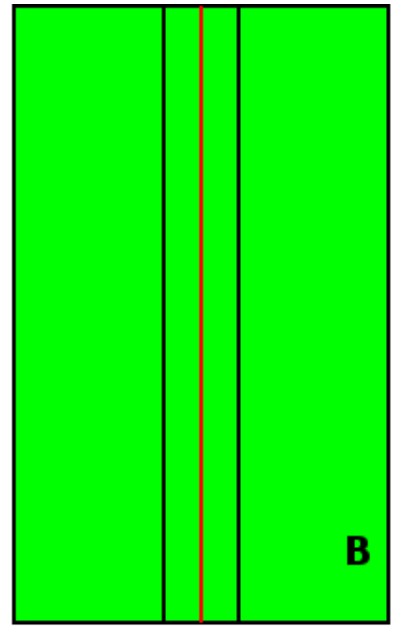
- compute one value of C by thread
- run on one row of A and one column of B by thread
- direct access from/to global memory
- kernel very similar to CPU code

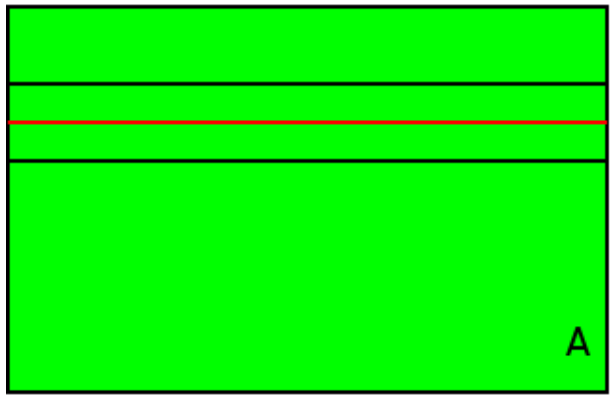
Why?

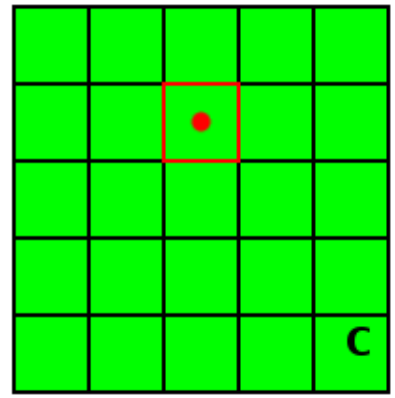
- easy-to-write
- very simple (maybe too simple?)

Parameter of the kernel:

- dimGrid = (m/32, n/32, 1)
- dimBlock = (32, 32, 1)







Simple-minded implementation (2/2)

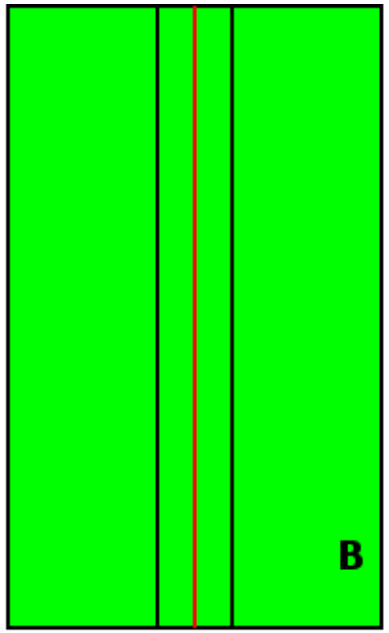
Source Code in CUDA C:

```
 __global__
void MatrixMulKernel
(float * Md, float * Nd, float * Pd, int Width)
{

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

   float Pvaleur = 0;
   float MdElement , NdElement;
   for (i = 0; i < Width; i++)
    {
       MdElement = Md[threadIdx.y * Width + i];
       NdElement = Nd[i  * Width + threadIdx.x];
       Pvaleur+=MdElement*NdElement;
    }

    Pd[ty * Width + tx] = Pvaleur;

}
```

Source Code in CUDA Fortran:

```
attributes(global) subroutine matrix_mul(a, b, c, m, l, n)
  implicit none
  real(fp_kind), intent(in) :: a(l,m), b(l,n)
  real(fp_kind), intent(out) :: c(m,n)
  integer, value :: m, l, n
  real(fp_kind) :: cij
  integer :: k, ix, iy

  ix = (blockIdx%x-1) * dimBlock%x + threadIdx%x
  iy = (blockIdx%y-1) * dimBlock%y + threadIdx%y

  cij = 0.0
  do k=1,l
    cij = cij + a(k,ix)*b(k,iy)
  enddo

  c(ix,iy) = cij
end subroutine matrix_mul
```

Use shared memory for matrix A (1/2)

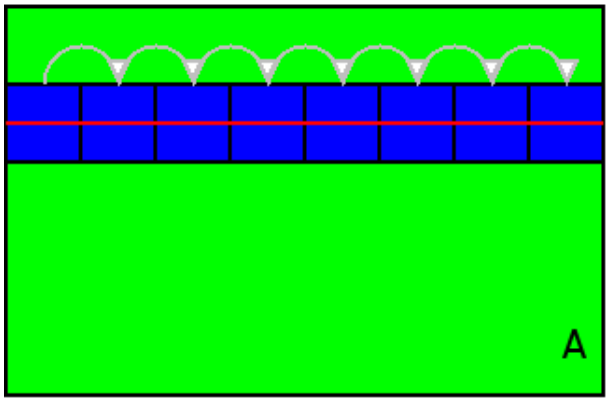


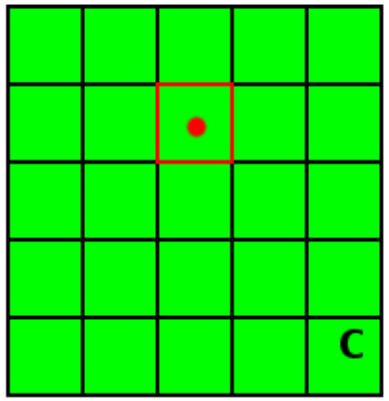
Idea:

- compute one value of C by thread
- run on one row of A and one column of B by thread
- store parts of rows of A in shared memory
- have to iterate to put all the row of A in shared memory
- direct access from global memory for B

Why?

- shared memory have lower latency
- shared memory have higher bandwidth
- more control on access memory





Parameter of the kernel:

- dimGrid = (m/32, n/32, 1)
- dimBlock = (32, 32, 1)

Use shared memory for matrix A (2/2)

Source Code in CUDA C:

```
 __global__
void MatrixMulKernel
(float * Md, float * Nd, float * Pd, int Width)
{
   int i;
   __shared__ float aTile[32][32];

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

   float Pvaleur = 0;
   float MdElement , NdElement;
   for (i = 0; i < Width; i++)
    {
        NdElement = Nd[i  * Width + threadIdx.x];
       Pvaleur+=aTile[tx][ty]*NdElement;
    }

  Pd[ty * Width + tx] = Pvaleur;
}
```

Source Code in CUDA Fortran:

```
attributes(global) subroutine matrix_mul(a, b, c, m, l, n)
  implicit none
  real, intent(in) :: a(l,m), b(l,n)
  real, intent(out) :: c(m,n)
  integer, value :: m, l, n
  real, shared :: a_shared(32, 32)
  real :: cij
  integer :: i, j, ix, iy, tx, ty

  tx  = threadIdx%x ; ty  = threadIdx%y
  ix = (blockIdx%x-1) * dimGrid%x + tx
  iy = (blockIdx%y-1) * dimGrid%y + ty

  cij = 0.0
  do i = 1, l, 32
    a_shared(tx,ty) = a(i+ty-1, ix)
    call syncthreads()
    do j = 1, 32
      cij = cij + a_shared(tx,j)*b(i+j-1, iy)
    enddo
    call syncthreads()
  enddo
  c(ix,iy) = cij
end subroutine matrix_mul
```

Use shared memory for matrix A and B (1/2)

<u>Idea:</u>

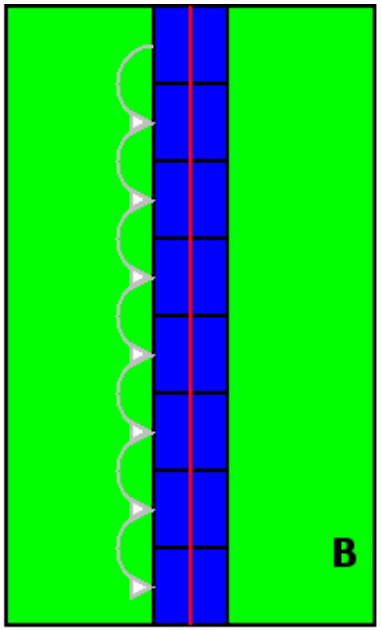
- compute one value of C by thread
- run on one row of A and one column of B by thread
- store parts of rows of A and B in shared memory
- have to iterate to put all the row of A and B in shared memory
- increase number of iterations

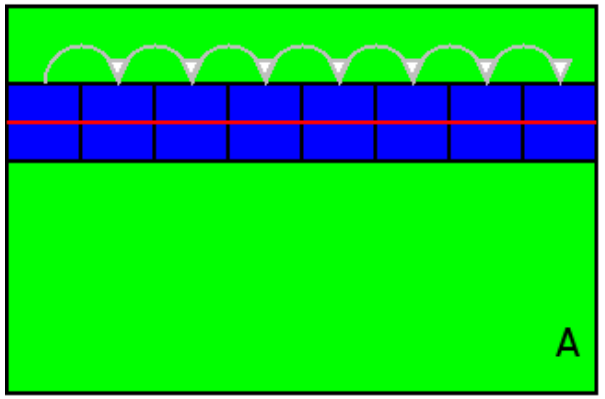
<u>Why?</u>

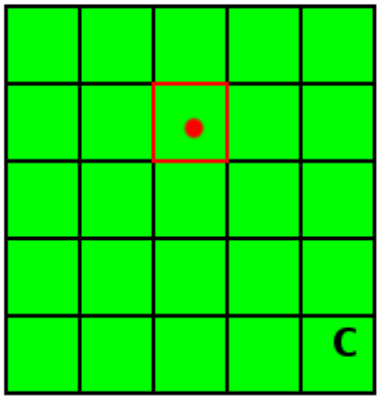
- shared memory have lower latency
- shared memory have higher bandwidth
- more control on access memory

<u>Parameter of the kernel:</u>

- dimGrid = (m/16, n/16, 1)
- dimBlock = (16, 16, 1)





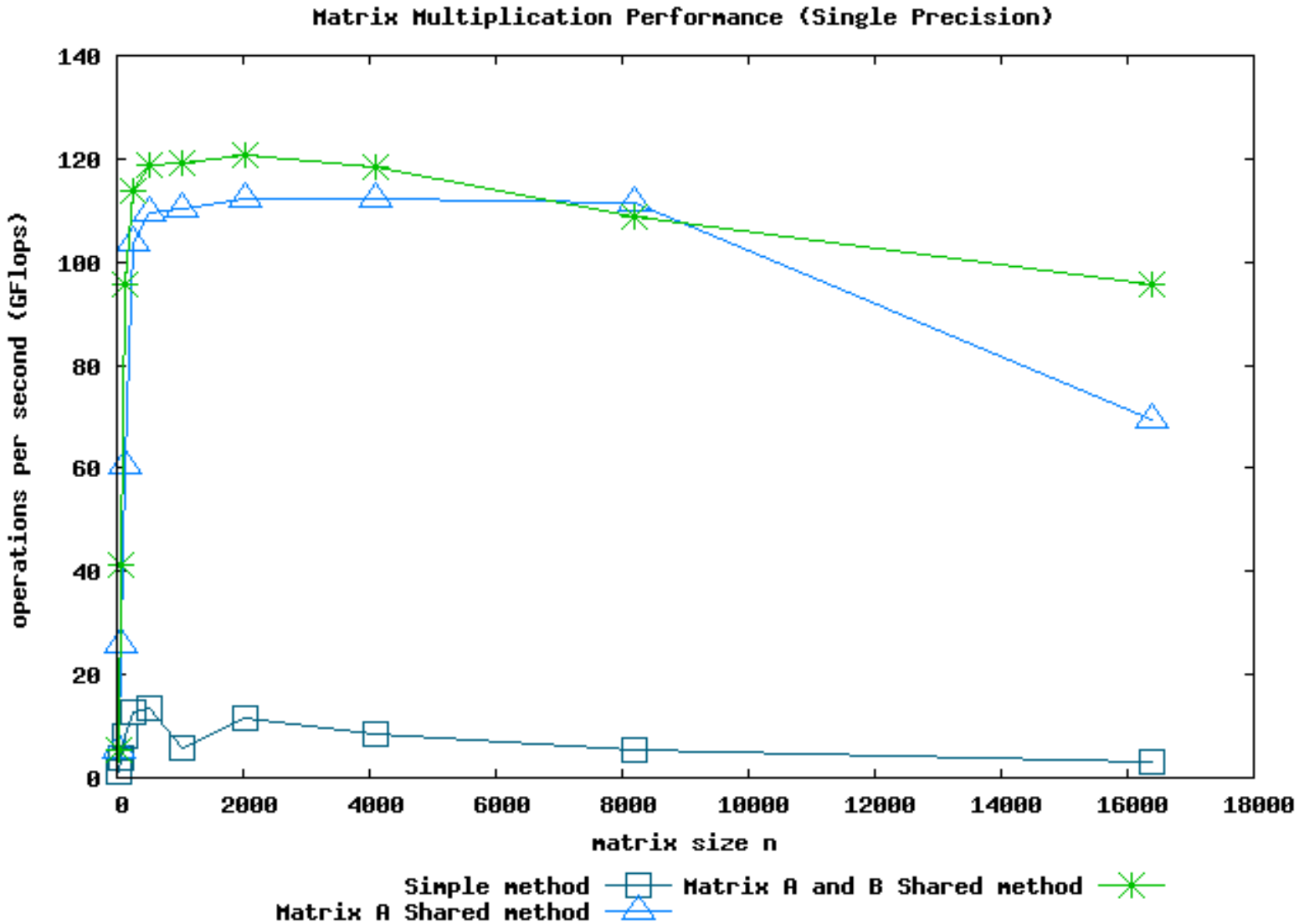


Use shared memory for matrix A and B (2/2)

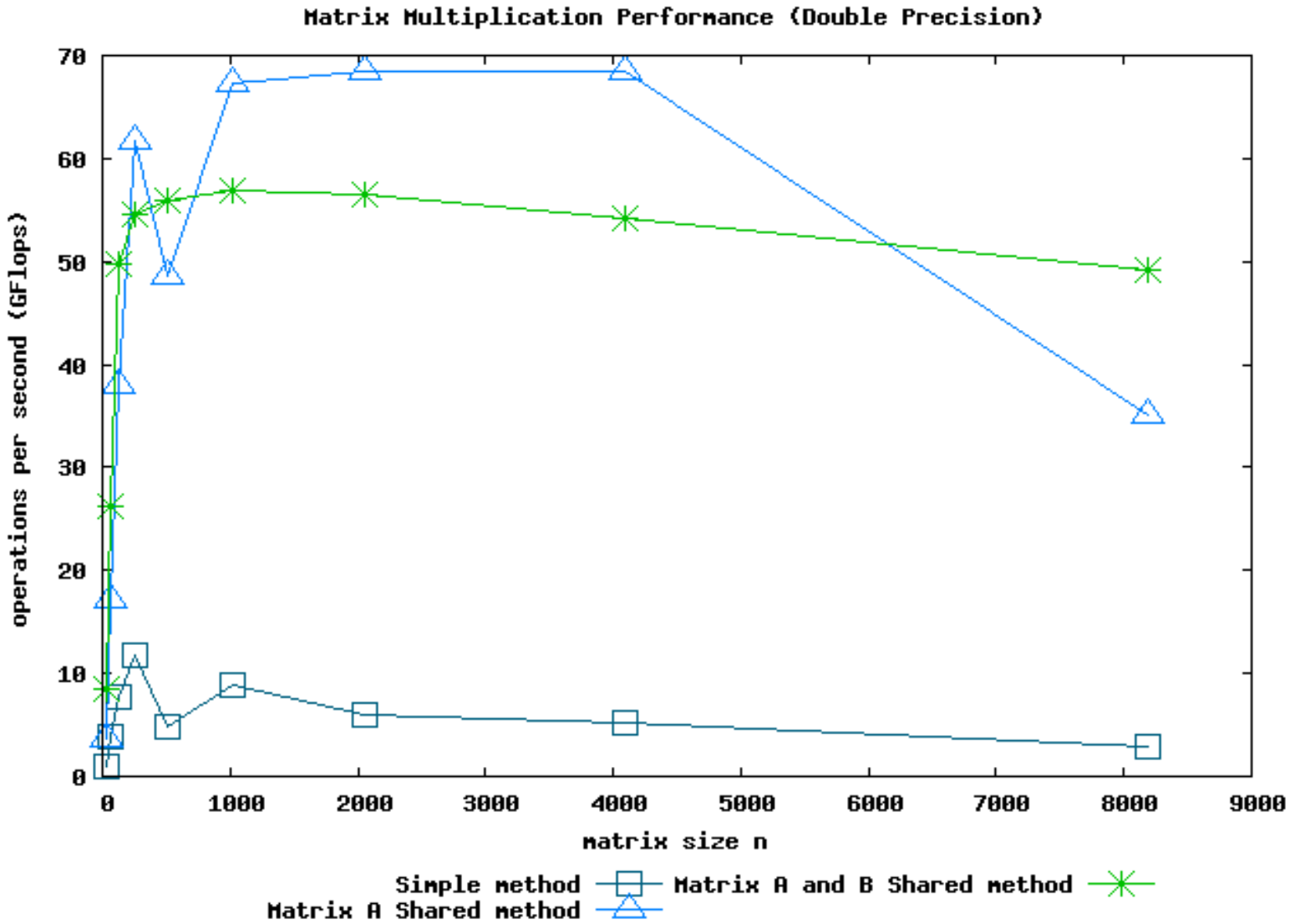
Source Code in CUDA Fortran:

```
attributes(global) subroutine matrix_mul(a, b, c, m, l,
n)
  implicit none
  real(fp_kind), intent(in) :: a(l,m), b(l,n)
  real(fp_kind), intent(out) :: c(m,n)
  integer, value :: m, l, n
  real(fp_kind), shared :: a_shared(16, 16)
  real(fp_kind), shared :: b_shared(16, 16)
  real(fp_kind) :: cij
  integer :: i, j, ix, iy, tx, ty

  tx  = threadIdx%x
  ty  = threadIdx%y
  ix = (blockIdx%x-1) * blockDim%x + tx
  iy = (blockIdx%y-1) * blockDim%y + ty

  cij = 0.0

  do i = 1, l, 32

    a_shared(tx,ty) = a(i+ty-1, ix)
    b_shared(tx,ty) = b(i+tx-1, iy)
    call syncthreads()

    do j = 1, 32
      cij = cij + a_shared(tx,j)*b_shared(j,ty)
    enddo
    call syncthreads()

  enddo
  c(ix,iy) = cij

end subroutine matrix_mul
```

Which performance in the end?



Which performance in the end?



Help me! Where are the libraries?

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
BORDEAUX - SUD-OUEST

Libraries for CUDA?

- CUBLAS
  - set of basic linear algebra routines
  - implementation of BLAS on top of CUDA Runtime
- CUFFT
  - used for data signal processing and solving partial differential equations
  - simple interface for FFT computation on GPU
- NPP
  - used for imaging and video processing
  - like implementation of IPP (Intel) on GPU
- CUSP
  - library for sparse linear algebra and graph computations
  - opensource project on Google Code
- CULA
  - library for solving systems of simultaneous linear equations
  - based-on LAPACK
  - EM Photonics in partnership with NVIDIA

Do not you see BLAS right there?

- (CU)BLAS is an organized library for basic linear algebra in 3 levels:
  - BLAS-1: operations between vectors
  - BLAS-2: operations between a matrix and a vector
  - BALS-3 : operations between matrices
- Our interests : <t>GEMM
  - function performs the matrix-matrix multiplication:

$$C = \alpha op(A) op(B) + \beta C$$

where α and β are scalars,
A, B, C are matrices,
and op() is (conjugate) transpose or not

CUBLAS help to program quickly (1/5)

Source Code in CUDA C:

```
 int main(int argc, char * argv[])
{
int i;
int Width;
int iteration;

double *M;
double *N;
double *P;

void **Md;
void **Nd;
void **Pd;

M = (double*) malloc (size * sizeof(double));
N = (double*) malloc (size * sizeof(double));
P = (double*) malloc (size * sizeof(double));

cublasInit();
cublasAlloc( size , sizeof(double) , (void**)&Md);
cublasSetMatrix(Width,Width,sizeof(double),M,Width,Md,
Width);
cublasAlloc( size , sizeof(double) , (void**)&Nd);
cublasSetMatrix(Width,Width,sizeof(double),N,Width,Nd,
Width);

cublasAlloc( size , sizeof(double) ,(void**)&Pd);

cublasDgemm ('N','N',Width,Width,Width,(double) 1.0,(const
double *)Md,Width,(const double *)Nd,Width,(double) 0.0,
(double *)Pd,Width);

cublasGetMatrix(Width,Width,sizeof(double),Pd,Width,P,
Width);

cublasFree(Md);
cublasFree(Nd);
cublasFree(Pd);

free(M);
free(N);
free(P);

}
```

CUBLAS help to program quickly (2/5)

Options in CUDA Fortran:

- ISO_C_BINDING:
  - binding NVIDIA CUDA C functions to define subroutine
  - using PGI extensions for memory management
- NON Thunking method:
  - using NVIDIA interface with manual memory management
  - using cublasAlloc / cublasFree
  - using cublasSetMatrix / cublasGetMatrix
  - using cublasSGEMM / cublasDGEMM
- Thunking method:
  - using NVIDIA interface with automaticl memory management
  - not need to alloc GPU memory
  - using cublasSGEMM / cublasDGEMM only

CUBLAS help to program quickly (3/5)

ISO C Binding method:

```
interface cuda_gemm
subroutine cuda_sgemm(cta, ctb, m, n, k,alpha,
A, lda, B, ldb, beta, c, ldc) bind(C,
name='cublasSgemm')
   use iso_c_binding
   character(1,c_char),value :: cta, ctb
   integer(c_int),value :: m,n,k,lda,ldb,ldc
   real(c_float),value :: alpha,beta
   real(c_float), device, dimension(lda,*) :: A
   real(c_float), device, dimension(ldb,*) :: B
   real(c_float), device, dimension(ldc,*) :: C
end subroutine cuda_sgemm
end interface cuda_gemm

program cublas_iso_c_binding
    use cudafor
    implicit none
    real, allocatable :: a(:,:), b(:,:), c(:,:)
    real, allocatable, device :: a_d(:,:), b_d(:,:), c_d(:,:)
    real(fp_kind) :: alpha, beta
    integer :: m, l, n
    integer :: i, j, k

    m = 1024 ; l = 1024 ; n = 1024
    allocate (a(m,l), b(l,n), c(m,n))

    a = 1.0 ; b = 2.0 ; c = 3.0
    alpha = 1.0 ; beta  = 0.0

    allocate (a_d(m,l), b_d(l,n), c_d(m,n))
    a_d = a ; b_d = b

    call cuda_gemm ('N','N',m,n,l,alpha,a_d,m,b_d,l,
beta,c_d,m)
    c = c_d

    deallocate (a,b,c)
    deallocate (a_d, b_d, c_d)
end program cublas_iso_c_binding
```

CUBLAS help to program quickly (4/5)

Non thunking method:

```
program cublas_non_thunking
   implicit none
   real, allocatable :: a(:,:), b(:,:), c(:,:)
   type(c_ptr) :: a_d, b_d, c_d
   real :: alpha, beta
   integer :: m, l, n, fp_kind=4

   m = 1024 ; l = 1024 ; n = 1024
   allocate (a(m,l), b(l,n), c(m,n))

   a = 1.0 ; b = 2.0 ; c = 3.0
   alpha = 1.0 ; beta  = 0.0

   call cublas_Init()
   call cublas_Alloc(m*l, fp_kind, a_d)
   call cublas_Alloc(l*n, fp_kind, b_d)
   call cublas_Alloc(m*n, fp_kind, c_d)
   call cublas_Set_Matrix(m, l, fp_kind, a, l, a_d, l)
   call cublas_Set_Matrix(l, n, fp_kind, b, n, b_d, n)

   call cublas_SGEMM('n', 'n', m, n, l, alpha, a_d, l,
b_d, n, beta, c_d, n)
   call cublas_Get_Matrix(m, n, fp_kind, c_d, n, c, n)

   deallocate (a,b,c)
   call cublas_Free(a_d)
   call cublas_Free(b_d)
   call cublas_Free(c_d)
   call cublas_Shutdown()

end program cublas_non_thunking
```

CUBLAS help to program quickly (5/5)

Non thunking method:

```
program cublas_thunking
    implicit none
    real, allocatable :: a(:,:), b(:,:), c(:,:)
    real :: alpha, beta
    integer :: m, l, n

    m = 1024 ; l = 1024 ; n = 1024
    allocate (a(m,l), b(l,n), c(m,n))

    a = 1.0 ; b = 2.0 ; c = 3.0
    alpha = 1.0 ; beta  = 0.0

    call cublas_SGEMM('n', 'n', m, n, l, alpha, a, l, b, n, beta, c, n)

    deallocate (a,b,c)
end program cublas_thunking
```

Compilation

CUDA C with CuBLAS:

```
module load compiler/gcc gpu/cuda/3.2.16
nvcc *.cu -lcudart -o prog
```

ISO C Binding:

```
module load compiler/pgi gpu/cuda/3.2.16
pgfortran -c m_interface.f90
pgfortran -fast -Mcuda=3.2 -lcublas m_interface.o cublas_iso_c_binding.cuf
```

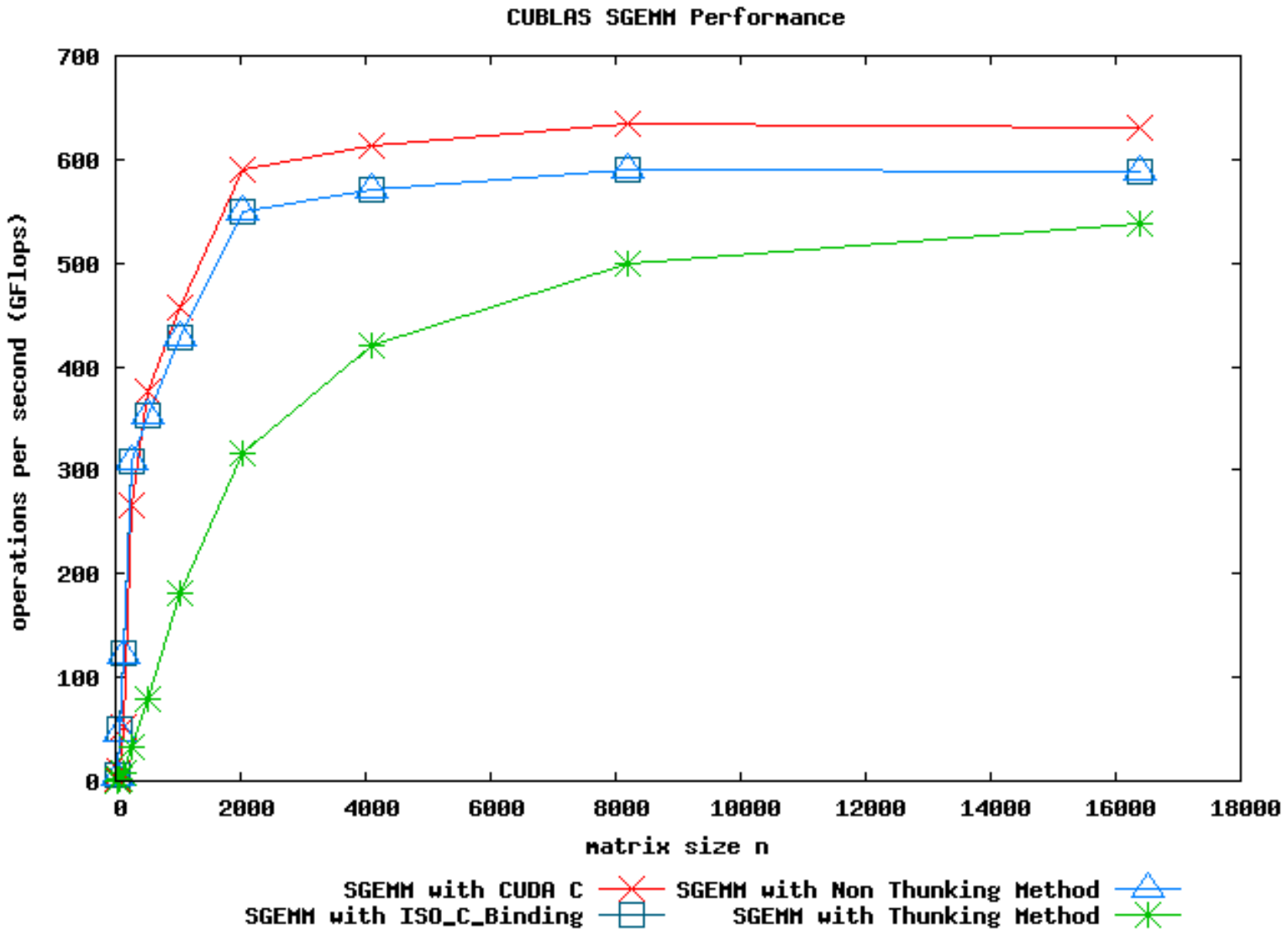
Thunking method:

```
module load compiler/pgi gpu/cuda/3.2.16
gcc -O3 -I$(CUDA_TOOLKIT)/include -c $(CUDA_TOOLKIT)/src/fortran_thunking.c
pgfortran -fast -Mcuda=3.2 -lcublas fortran_thunking.o cublas_thunking.cuf
```

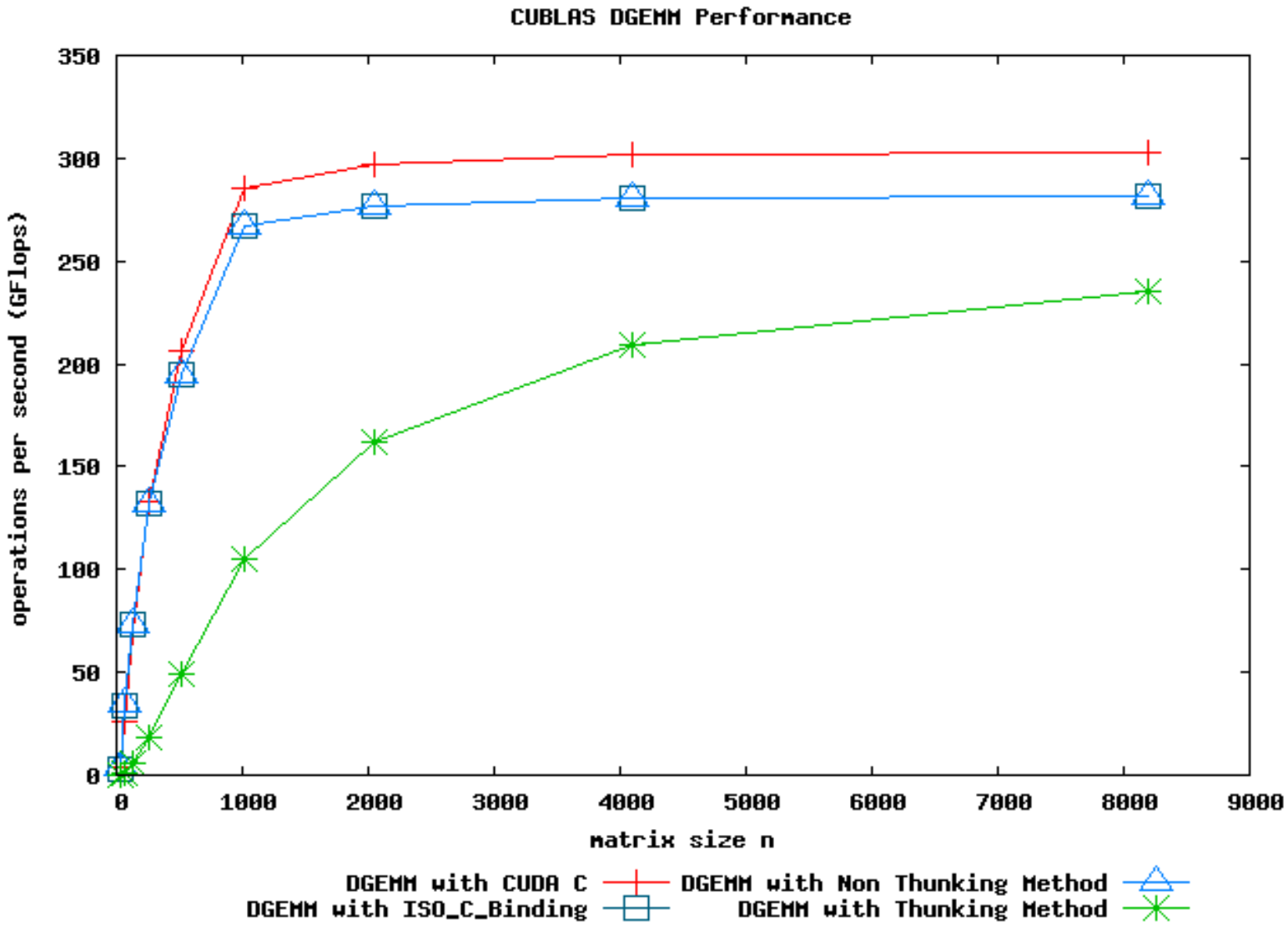
Non thunking method:

```
module load compiler/pgi gpu/cuda/3.2.16
gcc -O3 -I$(CUDA_TOOLKIT)/include -c $(CUDA_TOOLKIT)/src/fortran.c
pgfortran -fast -Mcuda=3.2 -lcublas fortran.o cublas_non_thunking.cuf
```

CUBLAS help to find performance



CUBLAS help to find performance



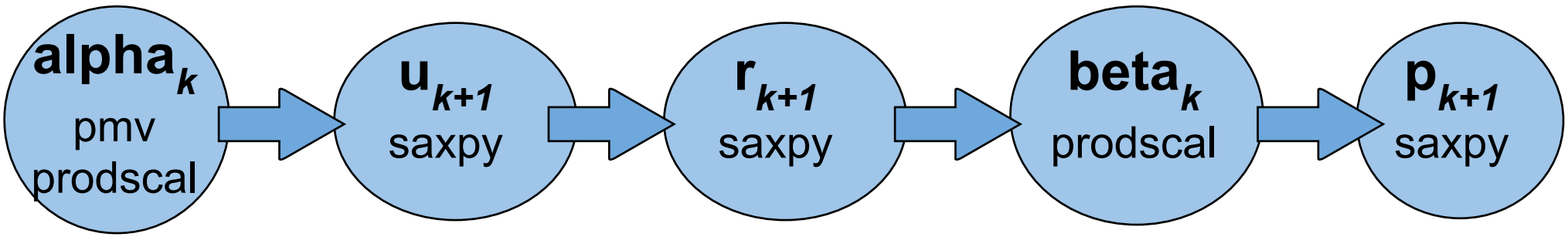
Outcome of all these sheets

- Use of libraries provides performance
  - over twice for matrix multiplication at least
- Simplify development of applications
  - focus on the integration of library calls when it is possible
  - develop kernel only if it is necessary
- Try to figure out when you have to call GPU implementation
  - bandwidth host <==> device is significant
  - give you indicators for GPU callings
- Give time to focus on optimizing other parts of program
  - profile your code to find hotspots
  - try to evaluate the necessity of GPU portability

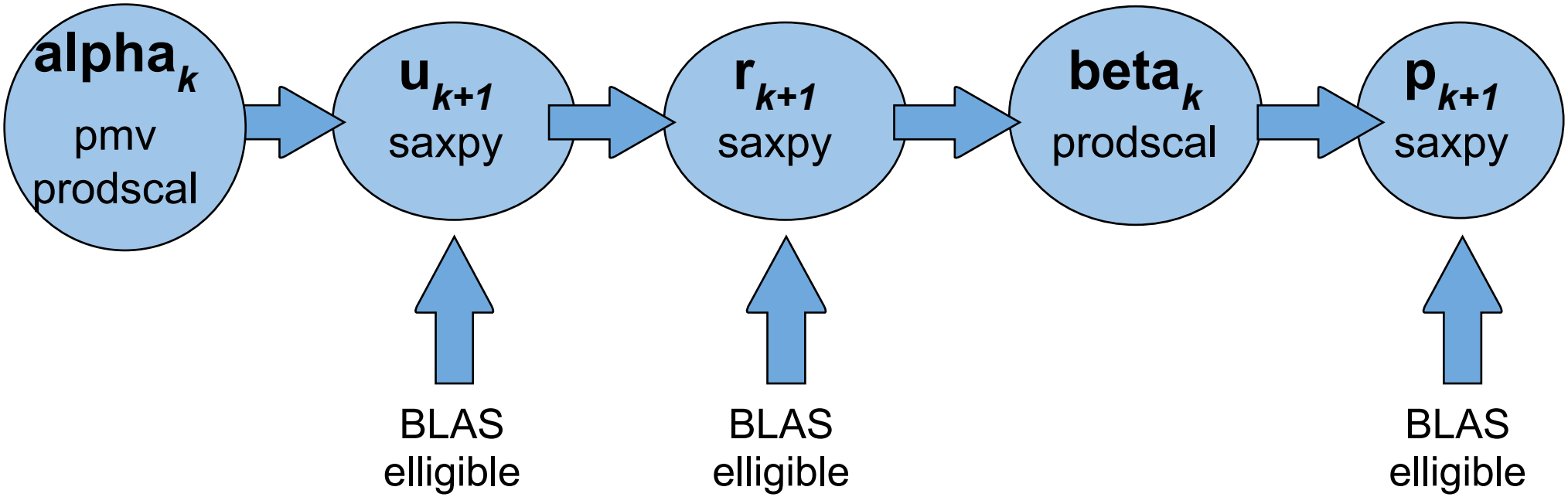
And in real life...

You have a CPU implementation

- Poisson Equation
  - conjugate gradient method
  - finite difference method
  - Jacobi solver



Find out where you can place libraries calls



Use Blas performance in this code ?

What we need to run on GPU?

```
do (it = 0 ; it < 1000 ; it ++)

   //calcul de \alpha_k
   (...)
   pmv(q,p)
   prodscal(alpha_k,q,p,comm3d)
   (...)
   //calcul de u_k+1
   saxpy( u, 1.0,  p,    alpha_k )
   //calcul de r_k+1
   saxpy( r,  1.0,  q,  - alpha_k )
   // calcul de \beta_k
   prodscal(  rnorm_k,  r,  r,  comm3d  )
   (...)
   // calcul de p_k+1
   saxpy(  p,  beta_k,  r,  1.0 )
   convergence()

enddo
```

Fortran thunking mode

→ cublas_daxpy(dimtot,alpha$_k$,p,1,u,1)

→ cublas_daxpy(dimtot,-alpha$_k$,q,1,r,1)

→ cublas_daxpy(dimtot,beta$_k$,p,1,r,1)

Outline

```
do (it = 0 ; it < 1000 ; it ++)

   //calcul de \alpha_k
   (...)
   pmv(q,p)
   prodscal(alpha_k,q,p,comm3d)
   (...)
   //calcul de u_k+1
   saxpy( u, 1.0,  p,    alpha_k )
   //calcul de r_k+1
   saxpy( r,  1.0,  q,  - alpha_k )
   // calcul de \beta_k
   prodscal(  rnorm_k,  r,  r,  comm3d  )
   (...)
   // calcul de p_k+1
   saxpy(  p,  beta_k,  r,  1.0 )
   convergence()

enddo
```

total time to compute :
- cpu implementation: 6.77 sec
- gpu implementation: 25.02 sec

Fortran thunking mode

→ cublas_daxpy(dimtot,alpha_k,p,1,u,1)

→ cublas_daxpy(dimtot,-alpha_k,q,1,r,1)

→ cublas_daxpy(dimtot,beta_k,p,1,r,1)

transfert :
11008 ko up&down
Bandwidth :
Peak perf 8Gb/s
time to transfert :
2,6 ms

added host/device bandwidth: 15,74 sec

Outlines

When going on GPU is profitable?

APPROACH	EXAMPLES
Application Integration	MATLAB, Mathematica, LabVIEW
Implicit Parallel Languages	PGI Accelerator, HMPP
Abstraction Layer/Wrapper	PyCUDA, CUDA.NET, jCUDA
Language Integration	CUDA C/C++, PGI CUDA Fortran
Low-level Device API	CUDA C/C++, DirectCompute, OpenCL

Go on GPU at your speed

- Profile your code or analyse your method
  - size of your problem
  - fine grain or coarse grain ?
  - be careful of the ratio data transfers versus kernel computation
- Needed time to have a portable code
  - often more than one year ...
  - dedicated to one kind of GPU architecture ?

- Extract some eligible kernels but ...
  - libraries do the work
  - libraries are not the solution for every problem

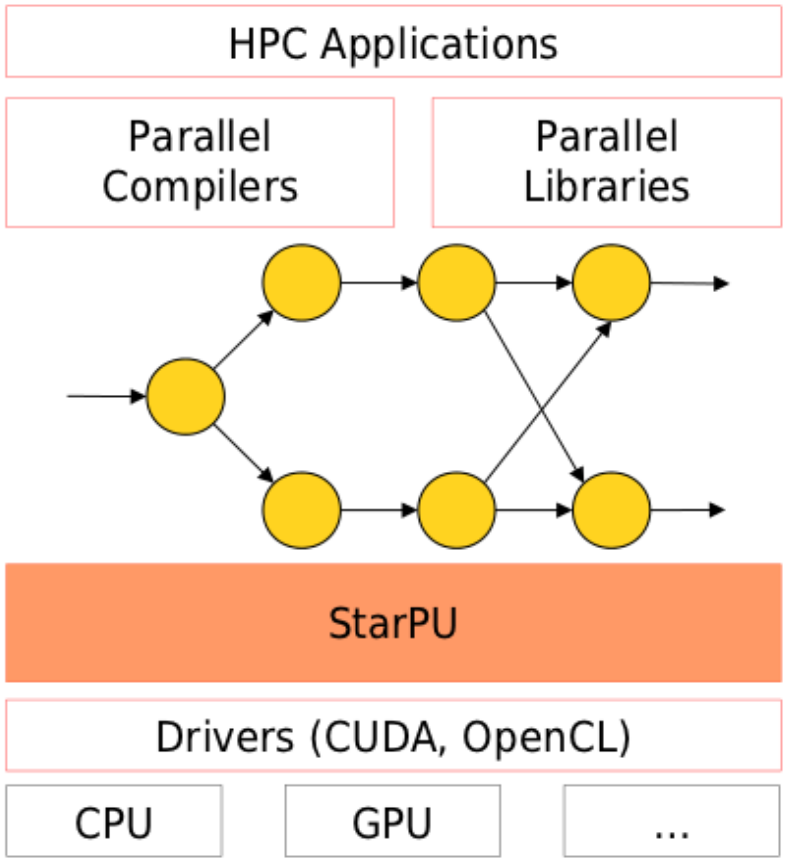
Your implementation will be portable?

- Adapt your kernels to card specifications:
  - type of memory
    - global memory,
    - shared memory,
    - texture memory...
  - size of memory
    - block size,
    - grid size...

- Choose your level
    - thunking or non-thunking
    - iso c-binding
    - low or high level API

What is your future?

- MPI + OpenMP + CUDA?
- Task scheduler and more ...
  - StarPU
  - RUNTIME Team
- PlaFRIM GT GPU :
  - gt-gpu@inria.fr
  - PlaFRIM Team



INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
BORDEAUX - SUD-OUEST

References

- http://developer.nvidia.com/category/zone/cuda-zone
- http://www.pgroup.com/resources/cudafortran.htm
- http://gpu4vision.icg.tugraz.at/
- http://gpgpu.org/
- http://tcuvelier.developpez.com/tutoriels/gpgpu/cuda/approfondi/?page=librairies