Computational Science Alliance
The University of Tokyo

Summer School 2018, Computational Science Alliance, UTokyo
Hands on Lecture on Aug. 22-24, 2018

GPU Computing by OpenACC・CUDA - 2nd day -

Masaharu MATSUMOTO
Project Lecturer, Department of Computer Science

matsumoto@is.s.u-tokyo.ac.jp

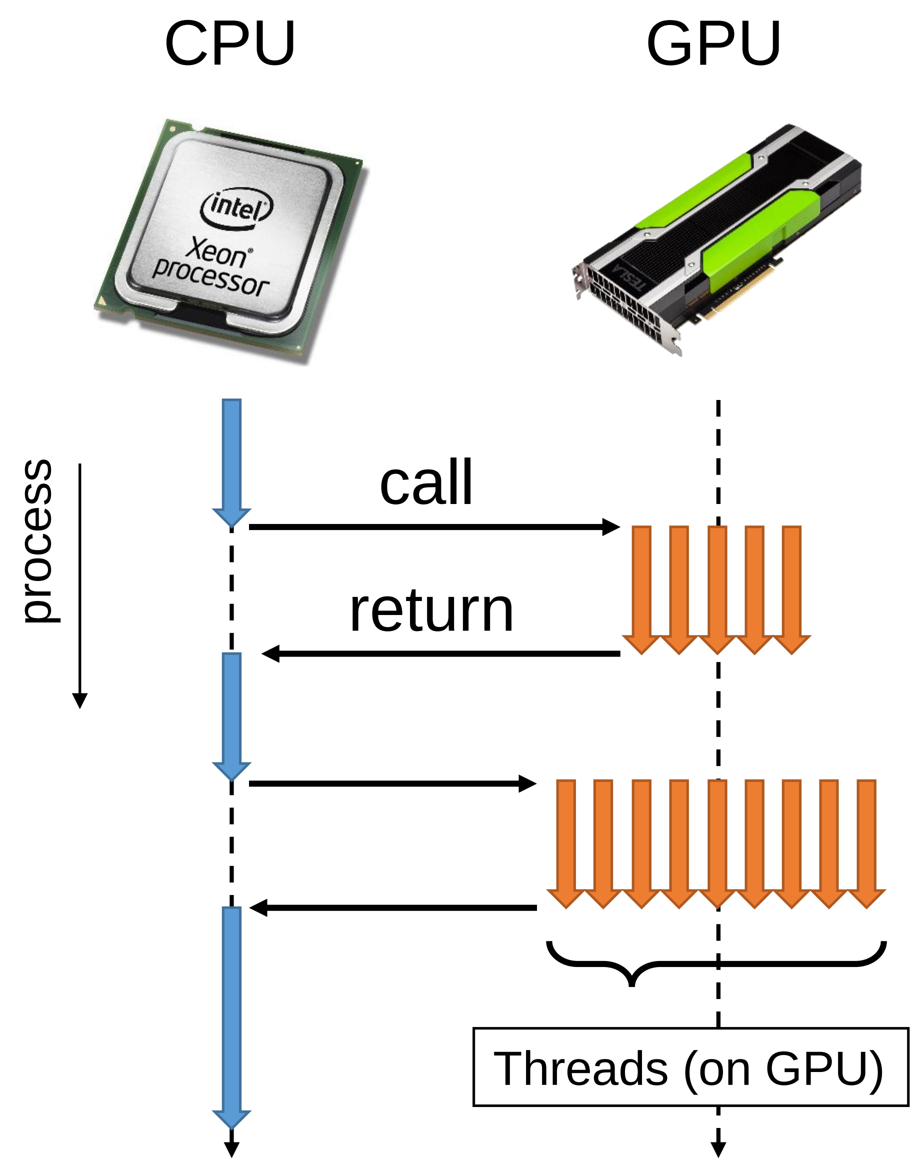
Programming Language for Accelerator

- OpenMP
  - ✓ Not only general CPU but also Intel Xeon Phi
- OpenACC
  - ✓ Directive-Based
  - ✓ Similar to OpenMP
- CUDA C/CUDA Fortran
  - ✓ Language for NVIDIA GPU
- OpenCL
  - ✓ NVIDIA GPU, AMD GPU, general multicore CPU, etc…
  - ✓ Coding is slightly complicated than CUDA

Programming Language CUDA

- ➢ Programming Language for NVIDIA GPU
  - ✓ Compute Unified Device Architecture
  - ✓ Jun. 2007, CUDA ver.1.0 release
  - ✓ ver.8 on Reedbush-H
  - ✓ Basically for 1GPU→CUDA+MPI for Multi-GPU
- ➢ Standard C subset + extension for GPGPU（CUDA C）, filename: hoge.cu
- ➢ Fortran ver. is provided by The Portland Group（CUDA Fortran）, filename: hoge.cuf

Programming model for CUDA



- Similar to OpenACC
- It starts from main function on CPU
  - ✓ GPU works when CPU submits a job processing
  - ✓ Function/Subroutine on GPU ＝ GPU kernel function
- CPU and GPU have respectively different memory space.
- Many threads work on GPU and the memory is shared.

CUDA Program Configulation

Host Function + GPU Kernel Function

- 2 types of functions(subroutines) in the source file
- Host function
  - ✓ Function executed on CPU
  - ✓ A processing is started from main function on C/C++, Fortran
  - ✓ Data transfer to GPU, Calling the GPU kernel
- GPU kernel function
  - ✓ Function executed on GPU
  - ✓ GPU kernel is executed by calling from Host fucntion
  - ✓ (simply) Kernel function

Sample Program (sumarray.cu)

＊Note that this code is useless and has no meaning to use GPU.

```
#include<stdio.h>
#include<stdlib.h>
#include<cuda.h>
#include<cuda_runtime.h>

__global__ void sum_array(int nemax,double *dA,
                          double *dB,double *dC){
  int ni;
  for(ni=0;ni<nemax;ni++){
    dC[ni]=dA[ni]+dB[ni];
  }
  return;
}

int main(int argc,char *argv[]){
  const int nemax=128;
  int ni;
  double A[nemax],B[nemax],C[nemax];
  double *dA,*dB,*dC;
  // -- set initial value --
  srand(248309);
  for(ni=0;ni<nemax;ni++){
    A[ni]=(double)rand()/RAND_MAX;
    B[ni]=(double)rand()/RAND_MAX;
  }
  // -- allocate arrays on device --
  cudaMalloc((double **)&dA,nemax*sizeof(double));
  cudaMalloc((double **)&dB,nemax*sizeof(double));
  cudaMalloc((double **)&dC,nemax*sizeof(double));
  // -- copy memories from host to device --
  cudaMemcpy(dA,A,nemax*sizeof(double),
             cudaMemcpyHostToDevice);
  cudaMemcpy(dB,B,nemax*sizeof(double),
             cudaMemcpyHostToDevice);
  // -- sum --
  sum_array<<<1,1>>>(nemax,dA,dB,dC);
  // -- copy memories from device to host --
  cudaMemcpy(C,dC,nemax*sizeof(double),
             cudaMemcpyDeviceToHost);
  // -- output --
  for(ni=0;ni<nemax;ni++){
    printf("%d: %lf + %lf = %lf¥n",
           ni,A[ni],B[ni],C[ni]);
  }
  // -- deallocate arrays on Device --
  cudaFree(dA); cudaFree(dB); cudaFree(dC);
  cudaDeviceReset();
  return 0;
}
```

Sample Program (sumarray.cuf)

＊Note that this code is useless and has no meaning to use GPU.

```
module cudakernel
contains
  attributes(global) subroutine sum_array(nemax,dA,dB,dC)
    implicit none
    integer,value,intent(in)::nemax
    double precision,intent(in)::dA(nemax),dB(nemax)
    double precision,intent(out)::dC(nemax)
    integer::ni
    do ni=1,nemax
       dC(ni)=dA(ni)+dB(ni)
    enddo
    return
  end subroutine sum_array
end module cudakernel

program main
  use cudafor
  use cudakernel
  implicit none
  integer,parameter::nemax=128
  integer::ni,ierr
  double precision::A(nemax),B(nemax),C(nemax)
  double precision,device,allocatable::dA(:),dB(:),dC(:)
  ! -- set initial value --
  call set_seed(248309)
  do ni=1,nemax
     call random_number(A(ni))
     call random_number(B(ni))
  enddo
  ! -- allocate arrays on device --
  allocate(dA(nemax))
  allocate(dB(nemax))
  allocate(dC(nemax))
  ! -- copy memories from host to device --
  dA=A
  dB=B
  ! -- sum --
  call sum_array<<<1,1>>>(nemax,dA,dB,dC)
  ! -- copy memories from device to host --
  C=dC
  ! -- output --
  do ni=1,nemax
     write(*,'(i5,":",f10.7," +",f10.7," =",f10.7)')&
           ni,A(ni),B(ni),C(ni)
  enddo
  ! -- deallocate arrays on device --
  deallocate(dA); deallocate(dB); deallocate(dC)
  ierr=cudaDeviceReset()
end program main
```

Processing Flow of Typical CUDA Program

on CPU

1. Data region is allocated on GPU mem.

   ↓

2. Input data is transferred to GPU

   ↓

3. GPU kernel function is called

5. Output is returned to CPU mem.

on GPU

4. Kernel function is executed.

```
__global__ void func(…){
  …
  return;
}
```

C

```
attributes(global) subroutine func(…)
   …
   return
end subroutine func
```

Fortran

CPU (main) memory and GPU (device) memory are clearly distinguished.

Sample Program (sumarray.cu)

＊Note that this code is useless and has no meaning to use GPU.

```
#include<stdio.h>
#include<stdlib.h>
#include<cuda.h>
#include<cuda_runtime.h>

__global__ void sum_array(int nemax,double *dA,
                          double *dB,double *dC){
  int ni;
  for(ni=0;ni<nemax;ni++){
    dC[ni]=dA[ni]+dB[ni];
  }
  return;
}

int main(int argc,char *argv[]){
  const int nemax=128;
  int ni;
  double A[nemax],B[nemax],C[nemax];
  double *dA,*dB,*dC;
  // -- set initial value --
  srand(248309);
  for(ni=0;ni<nemax;ni++){
    A[ni]=(double)rand()/RAND_MAX;
    B[ni]=(double)rand()/RAND_MAX;
  }
```

Header file for CUDA

GPU kernel function

Initial data settings on CPU

```
// -- allocate arrays on device --
      lloc((double **)&dA,nemax*sizeof(double));
      lloc((double **)&dB,nemax*sizeof(double));
cudaMalloc((double **)&dC,nemax*sizeof(double));
// -- copy memories from host to device --
cudaMemcpy(dA,A,nemax*sizeof(double),
           cudaMemcpyHostToDevice);
                              ouble),
                              ice);

sum_array<<<1,1>>>(nemax,dA,dB,dC);
// -- copy memories from device to host --
cudaMemcpy(C,dC,nemax*sizeof(double),
           cudaMemcpyDeviceToHost);
// -- output --
for(ni=0;ni<nemax;ni++){
  printf("%d: %lf + %lf = %lf¥n",
         ni,A[ni],B[ni],C[ni]);
}
                             evice --
                           cudaFree(dC);

  return 0;
}
```

Sample Program（sumarray.cu）

＊Note that this code is useless and has no meaning to use GPU.

```
#include<stdio.h>
#include<stdlib.h>
#
#

__global__ void sum_array(int nemax,double *dA,

  int ni;
  for(ni=0;ni<nem
    dC[ni]=dA[ni]+d
  }
  return;
}

int main(int argc,char *ar
  const int nemax=128;
  int ni;
  double A[nemax],B[nemax],C[nemax];
  double *dA,*dB,*dC;
  // -- set initial value --
  sran
  for(
    A[
    B[ni]=(double)rand()/RAND_MAX;
  }
```

Data region on GPU is allocated

Data transfer to GPU

Call kernel function

Return output

Deallocate memory on GPU

```
  // -- allocate arrays on device --
  cudaMalloc((double **)&dA,nemax*sizeof(double));
  cudaMalloc((double **)&dB,nemax*sizeof(double));
  cudaMalloc((double **)&dC,nemax*sizeof(double));
  // -- copy memories from host to device --
  cudaMemcpy(dA,A,nemax*sizeof(double),
             cudaMemcpyHostToDevice);
  cudaMemcpy(dB,B,nemax*sizeof(double),
             cudaMemcpyHostToDevice);
  // -- sum --
  sum_array<<<1,1>>>(nemax,dA,dB,dC);
  // -- copy memories from device to host --
  cudaMemcpy(C,dC,nemax*sizeof(double),
             cudaMemcpyDeviceToHost);
  // -- output --
  for(ni=0;ni<nemax;ni++){
    printf("%d: %lf + %lf = %lf¥n",
           ni,A[ni],B[ni],C[ni]);
  }
  // -- deallocate arrays on Device --
  cudaFree(dA); cudaFree(dB); cudaFree(dC);
  cudaDeviceReset();
  return 0;
}
```

Sample Program (sumarray.cuf)

＊Note that this code is useless and has no meaning to use GPU.

```
module cudakernel
contains
  attributes(global) subroutine sum_array(nemax,dA,dB,dC)
    implicit none
    integer,value,intent(in)::nemax
    double precision,intent(in)::dA(nemax),dB(nemax)
    double precision,intent(out)::dC(nemax)
    integer::ni
    do ni=1,nemax
       dC(ni)=dA(ni)+dB(ni)
    enddo
    return
  end subroutine sum_array
end module cudakernel

program main
  use cudafor
  use cudakernel
  implicit none
  integer,parameter::nemax=128
  integer::ni,ierr
  double precision::A(nemax),B(nemax),C(nemax)
  double precision,device,allocatable::dA(:),dB(:),dC(:)
```

Header file for CUDA

Device attribute (Fortran only)

```
! -- set initial value --
call set_seed(248309)
do ni=1,nemax
   call random_number(A(ni))
   call random_number(B(ni))
enddo
! -- allocate arrays on device --
a
a
a
!                                              -
d
dB=B
! -- sum --
call sum_array<<<1,1>>>(nemax,dA,dB,dC)
! -- copy memories from device to host --
C=dC
! -- output --
do ni=1,nemax
   write(*,'(i5,":",f10.7," +",f10.7," =",f10.7)')&
         ni A(ni) B(ni),C(ni)

                             ays on device --
deallocate(dA); deallocate(dB); deallocate(dC)
ierr=cudaDeviceReset()
end program main
```

Initial data settings on CPU

GPU kernel function (must be in module)

Sample Program (sumarray.cuf)

＊Note that this code is useless and has no meaning to use GPU.

```
module cudakernel
contains
  attributes(global) subroutine sum_array(nemax,dA,dB,dC)
    implicit none
    integer,value,intent(in)::nemax

    integer::ni
    do ni=1,nemax
       dC(ni)=dA(ni)+
    enddo
    return
  end subroutine sum_
end module cudakernel

program main
  use cudafor
  use cudakernel
  implicit none
  integer,parameter::nemax=128
  integer::ni,ierr
  double p
  double p                                           (:)
```

Data region on GPU is allocated

Data transfer to GPU

Call kernel function

Return output

Deallocate memory on GPU

```
  ! -- set initial value --
  call set_seed(248309)
  do ni=1,nemax
     call random_number(A(ni))
     call random_number(B(ni))
  enddo
  ! -- allocate arrays on device --
  allocate(dA(nemax))
  allocate(dB(nemax))
  allocate(dC(nemax))
  ! -- copy memories from host to device --
  dA=A
  dB=B
  ! -- sum --
  call sum_array<<<1,1>>>(nemax,dA,dB,dC)
  ! -- copy memories from device to host --
  C=dC
  ! -- output --
  do ni=1,nemax
     write(*,'(i5,":",f10.7," +",f10.7," =",f10.7)')&
           ni,A(ni),B(ni),C(ni)
  enddo
  ! -- deallocate arrays on device --
  deallocate(dA); deallocate(dB); deallocate(dC)
  ierr=cudaDeviceReset()
end program main
```

Thread Configulation in CUDA

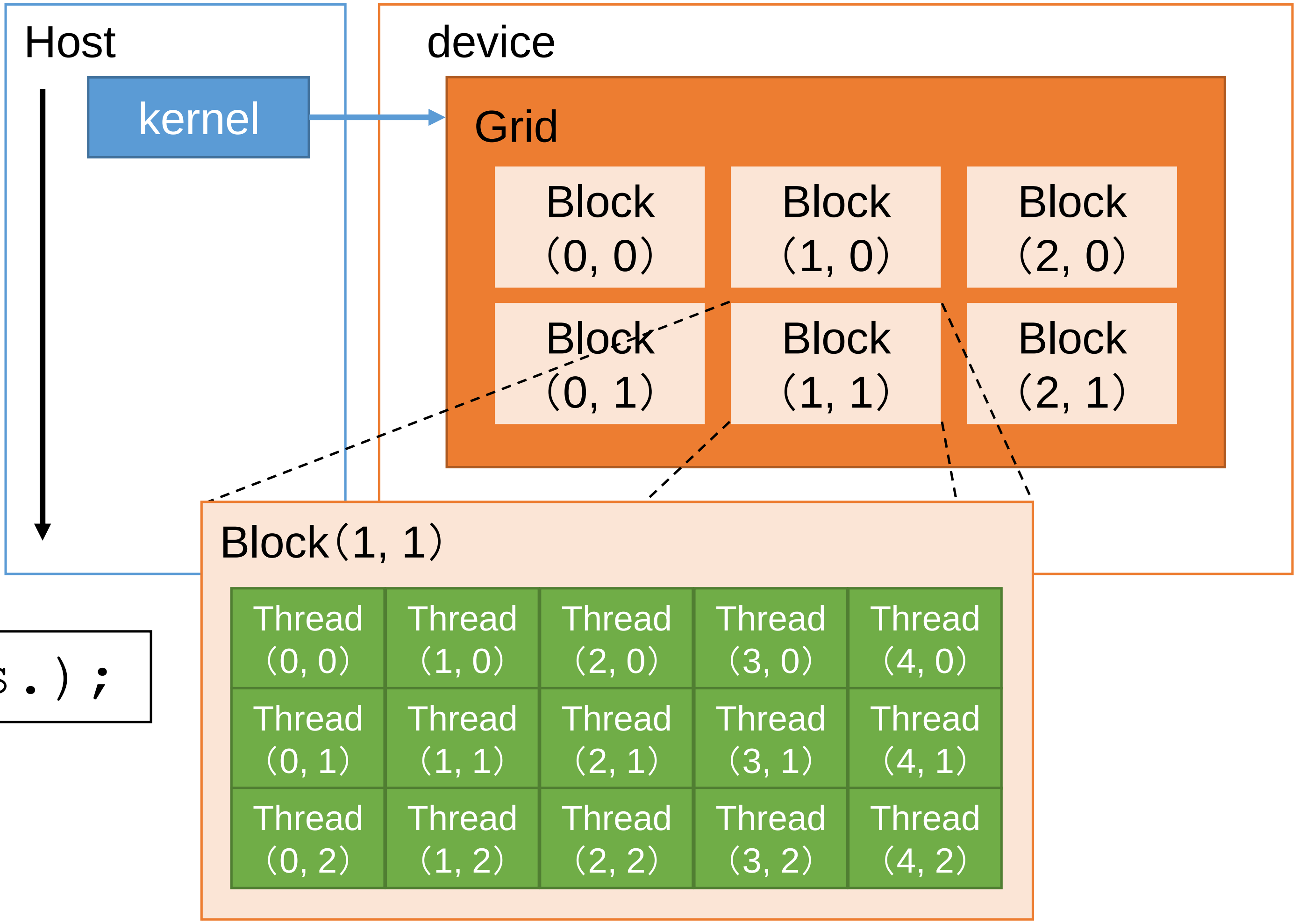
- Thread hierarchy is 2 level (Grid and Block) in CUDA.
- When kernel function is called, the number of threads is specified in 2 level (the number of blocks per grid and the number of threads per block).

```
func_kernel<<<30,128>>>(args.);
```

The number of blocks per grid

The number of threads per block

- `func_kernel` is executed by 30 x 128 = 3840 threads in the above example.



My Thread ID

- In a GPU kernel function, the following special variables can be obtained. Thereby my thread ID can be known although that is unknown in OpenACC.

- My ID（in C）
  - ✓ `blockIdx.x` How many blocks are there from 0
  - ✓ `threadIdx.x` How many threads are there from 0 in the block.

  （in Fortran, ″`.`″→″`%`″ and from 1）

- The number of threads（in C）
  - ✓ `gridDim.x` Total number of blocks
  - ✓ `blockDim.x` Total number of threads in each block

  （in Fortran, ″`.`″→″`%`″）

- A variable showing serial number is none.

  in C（from 0） → `blockIdx.x*blockDim.x+threadIdx.x`

  In Fortran（from 1） → `(blockIdx%x-1)*blockDim%x+threadIdx%x`

Improved Sample Program（CUDA C）

```
#include<stdio.h>
#include<stdlib.h>
#include<cuda.h>
#include<cuda_runtime.h>

__global__ void sum_array(int nemax,double *dA,
                          double *dB,double *dC){
  int ni;
  ni=blockIdx.x*blockDim.x+threadIdx.x;
  dC[ni]=dA[ni]+dB[ni];
  return;
}

int main(int argc,char *argv[]){
  const int nemax=128,thread=8;
  int ni;
  double A[nemax],B[nemax],C[nemax];
  double *dA,*dB,*dC;
  // -- set initial value --
  srand(248309);
  for(ni=0;ni<nemax;ni++){
    A[ni]=(double)rand()/RAND_MAX;
    B[ni]=(double)rand()/RAND_MAX;
  }
  // -- allocate arrays on device --
  cudaMalloc((double **)&dA,nemax*sizeof(double));
  cudaMalloc((double **)&dB,nemax*sizeof(double));
  cudaMalloc((double **)&dC,nemax*sizeof(double));
  // -- copy memories from host to device --
  cudaMemcpy(dA,A,nemax*sizeof(double),
             cudaMemcpyHostToDevice);
  cudaMemcpy(dB,B,nemax*sizeof(double),
             cudaMemcpyHostToDevice);
  // -- sum --
  sum_array<<<nemax/thread,thread>>>(nemax,dA,dB,dC);
  // -- copy memories from device to host --
  cudaMemcpy(C,dC,nemax*sizeof(double),
             cudaMemcpyDeviceToHost);
  // -- output --
  for(ni=0;ni<nemax;ni++){
    printf("%d: %lf + %lf = %lf¥n",
           ni,A[ni],B[ni],C[ni]);
  }
  // -- deallocate arrays on Device --
  cudaFree(dA); cudaFree(dB); cudaFree(dC);
  cudaDeviceReset();
  return 0;
}
```

Improved Sample Program（CUDA Fortran）

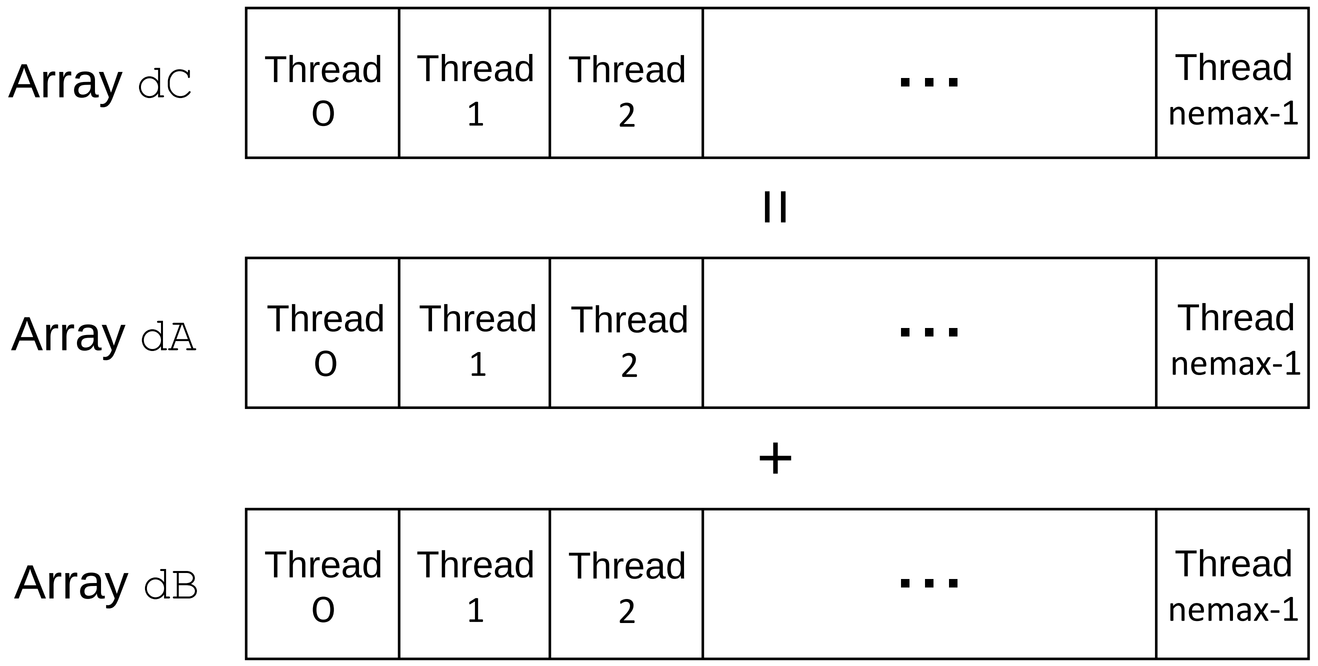
```fortran
module cudakernel
contains
  attributes(global) subroutine sum_array(nemax,dA,dB,dC)
    implicit none
    integer,value,intent(in)::nemax
    double precision,intent(in)::dA(nemax),dB(nemax)
    double precision,intent(out)::dC(nemax)
    integer::ni
    ni=(blockIdx%x-1)*blockDim%x+threadIdx%x
    dC(ni)=dA(ni)+dB(ni)
    return
  end subroutine sum_array
end module cudakernel

program main
  use cudafor
  use cudakernel
  implicit none
  integer,parameter::nemax=128,thread=8
  integer::ni,ierr
  double precision::A(nemax),B(nemax),C(nemax)
  double precision,device,allocatable::dA(:),dB(:),dC(:)
  ! -- set initial value --
  call set_seed(248309)
  do ni=1,nemax
     call random_number(A(ni))
     call random_number(B(ni))
  enddo
  ! -- allocate arrays on device --
  allocate(dA(nemax))
  allocate(dB(nemax))
  allocate(dC(nemax))
  ! -- copy memories from host to device --
  dA=A
  dB=B
  ! -- sum --
  call sum_array<<<nemax/thread,thread>>>(nemax,dA,dB,dC)
  ! -- copy memories from device to host --
  C=dC
  ! -- output --
  do ni=1,nemax
     write(*,'(i5,":",f10.7," +",f10.7," =",f10.7)')&
           ni,A(ni),B(ni),C(ni)
  enddo
  ! -- deallocate arrays on device --
  deallocate(dA); deallocate(dB); deallocate(dC)
  ierr=cudaDeviceReset()
end program main
```

Important Points in Improved Program (C)

- Thread ID 0 (in serial number) calculates `dC[0]`
- Thread ID 1 calculates `dC[1]`

⋮

- `nemax-1` th thread calculates `dC[nemax-1]`
- In order to obtain the serial number of thread, the following formula is calculated in each thread.



C: `ni=blockIdx.x*blockDim.x+threadIdx.x;`

Fortran: `ni=(blockIdx%x-1)*blockDim%x+threadIdx%x`

In Fortran, ID is from 1 to nemax

1 thread calculates 1 element "`dC[ni]`"

→ No for/do loop

→ More detailed description in programming than OpenACC

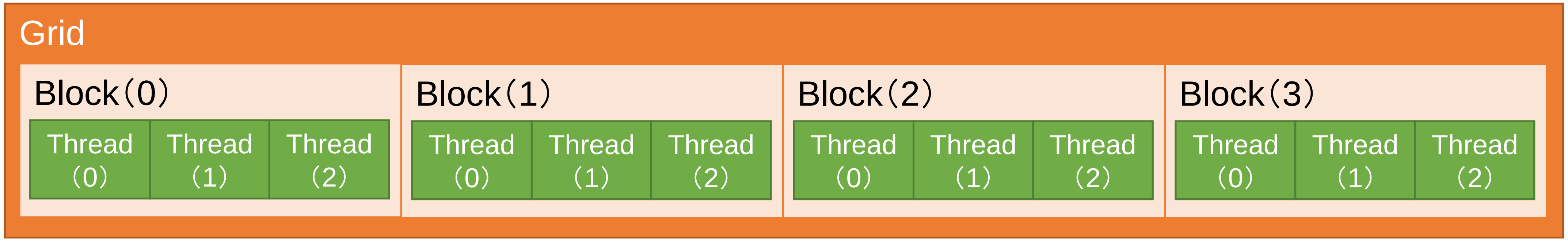
Multi-dimensional Blocks and Threads（1/2）

1 dimensional blocks and threads（C）

example：`kernel_func<<<4,3>>>();`

The number of blocks
`=gridDim`

The number of threads
`=blockDim`

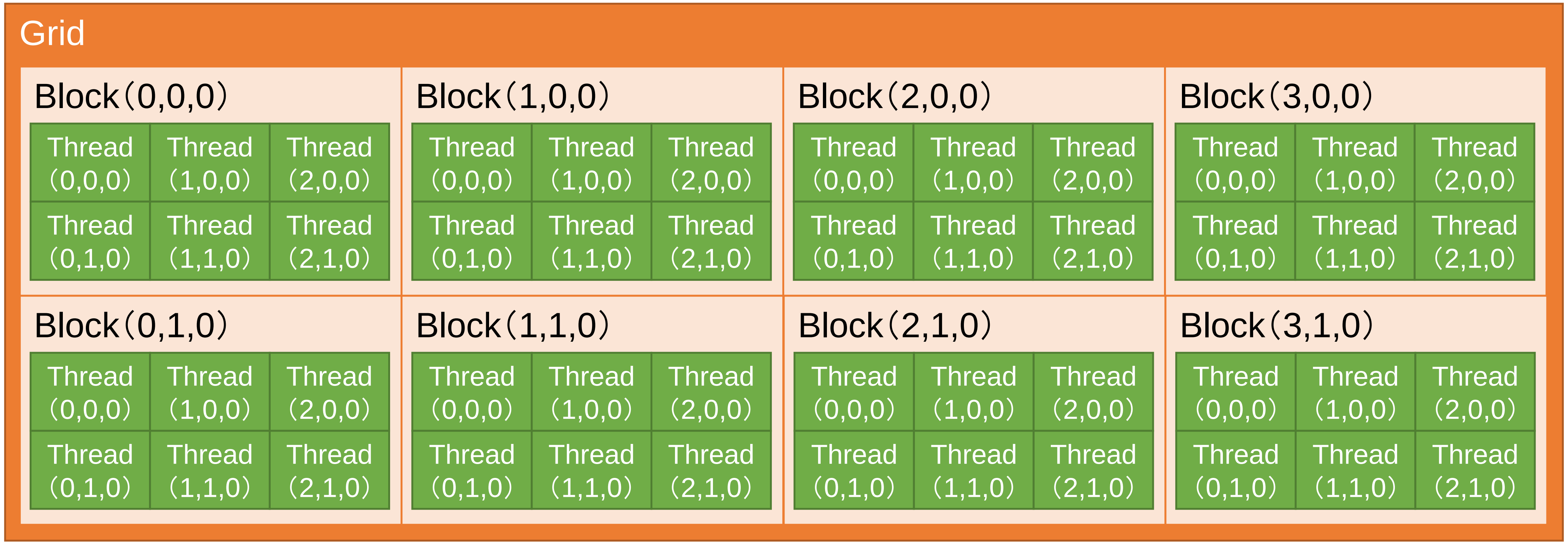


Multi-dimensional Blocks and Threads (2/2)

In order to specify the number of blocks and threads,

- `dim3` structure (3D of x, y, z)
- `int(integer)` type
  - `<<<a,b>>>` stands for `dim3(a,1,1),dim3(b,1,1)>>>`

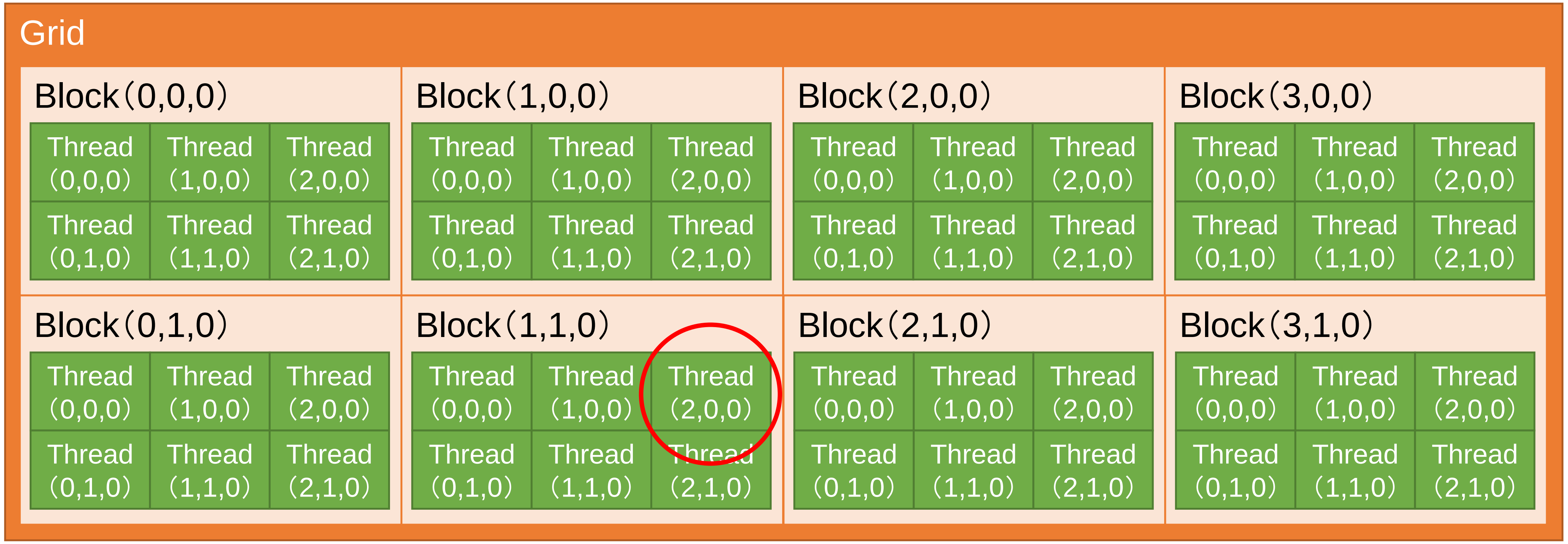
Example: `kernel_func<<<dim3(4,2,1),dim3(3,2,1)>>>();`



* `kernel_func<<<dim3(4,2),dim3(3,2)>>>();`, is also OK.

Thread ID in Multi-dimension (C)

```
kernel_func<<<dim3(4,2,1),dim3(3,2,1)>>>();
```



- In each thread,

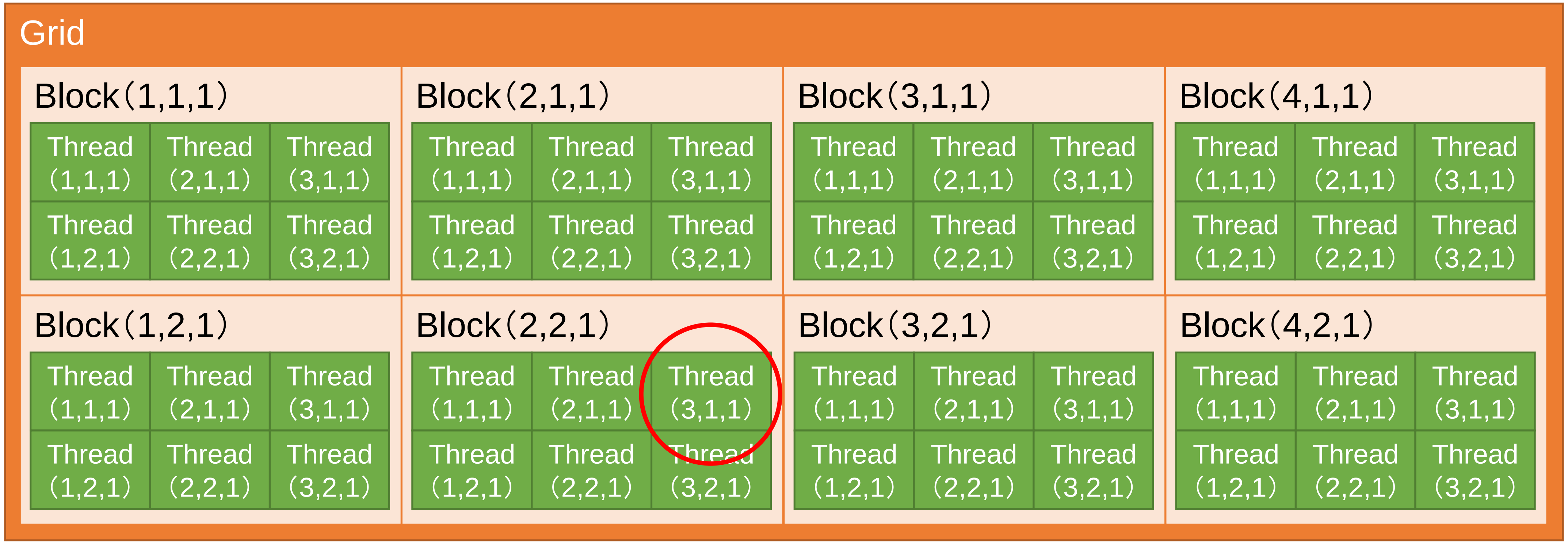
```
gridDim.x=4,  gridDim.y=2,  gridDim.z=1
blockDim.x=3, blockDim.y=2, blockDim.z=1
```

- From the thread of ○,

```
blockIdx.x=1, blockIdx.y=1, blockIdx.z=0
threadIdx.x=2,threadIdx.y=0,threadIdx.z=0
```

Thread ID in Multi-dimension（Fortran）

```
call kernel_func<<<dim3(4,2,1),dim3(3,2,1)>>>()
```



- In each thread,

```
gridDim.x=4,  gridDim.y=2,  gridDim.z=1
blockDim.x=3, blockDim.y=2, blockDim.z=1
```

- From the thread of ○,

```
blockIdx.x=2, blockIdx.y=2, blockIdx.z=1
threadIdx.x=3,threadIdx.y=1,threadIdx.z=1
```

Limitation of the Number of Blocks and Threads

- ➢ There is a limit to the number of blocks and threads.
- ➢ In the NVIDIA Tesla P100 in Reedbush,

  - ✓ Blocks ： x is max. 2^{31}-1, and y and z are max. 65535 respectively.
  - ✓ Threads： x and y are max. 1024 and z is max. 64, also total up to 1024

When the number of blocks and threads are specified, the number of threads is often fixed to max. 1024 and the number of blocks is increased.

  - ✓ The number of blocks and threads depend on GPU model

- ➢ Why is thread 2 level（Blocks and Threads) in CUDA ？

  - ✓ In order to correspond to hardware
    （In the case of P100, 1 GPU = 56 SM, 1 SM = 64 CUDA core）
  - ✓ The max number of threads becomes 2^{31}-1（2,147,483,647）×65,535×65,535 ×1,024. It is too large to specify by 1 level ID.

Examples of Thread configulation (C)

```
dim3 grid,block;
grid.x=2; grid.y=4;
block.x=8; block.y=16;
…
kernel_func<<<grid,block>>>(…);
```

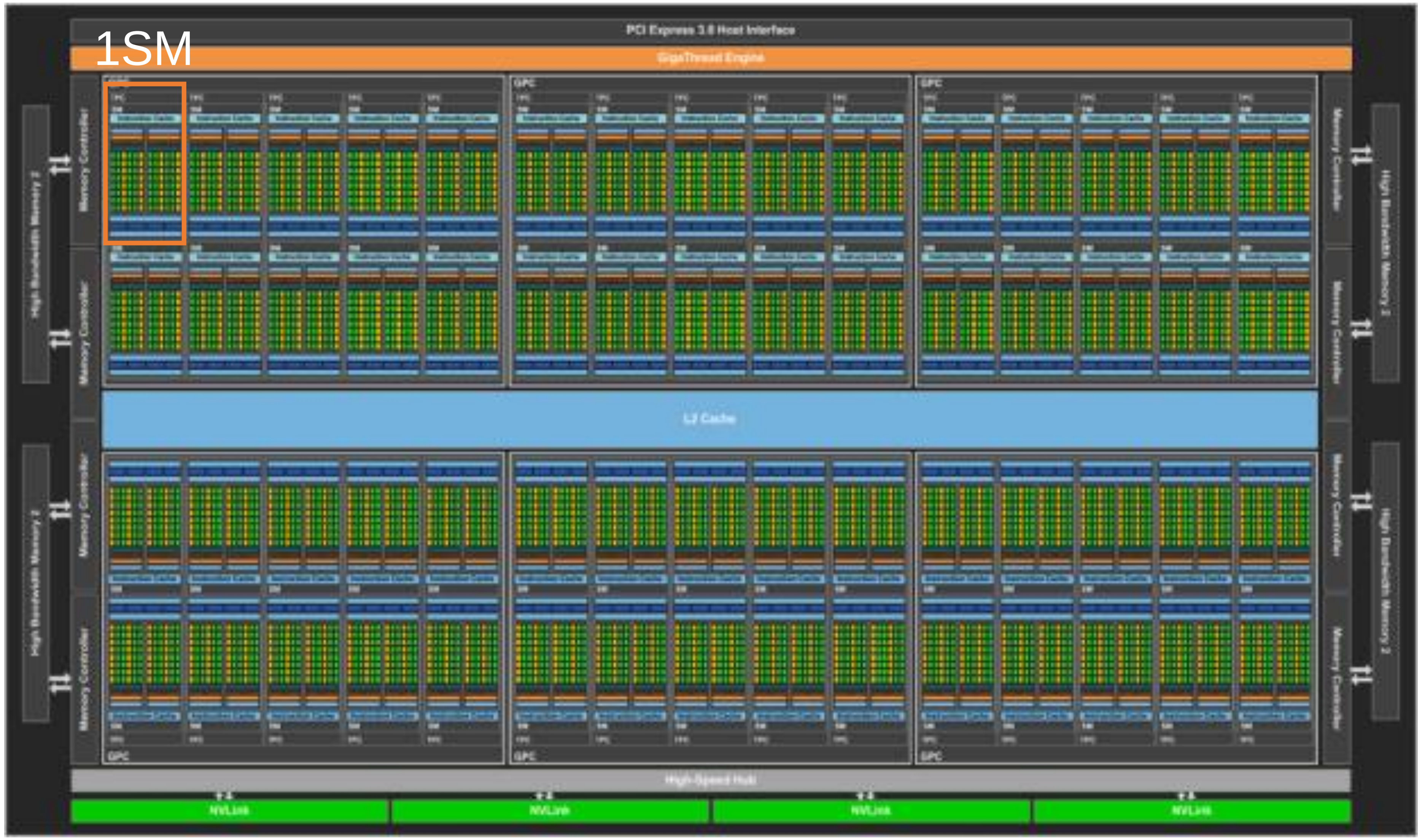
```
dim3 grid(2,4),block(8,16);
…
kernel_func<<<grid,block>>>(…);
```

```
…
kernel_func<<<32,512>>>(…);
```

The dimension specified by dim3 is unrelated to the performance.

GPU Architecture: NVIDIA Tesla P100

- 56 SMs
- 3584 CUDA Cores
- 16 GB HBM2

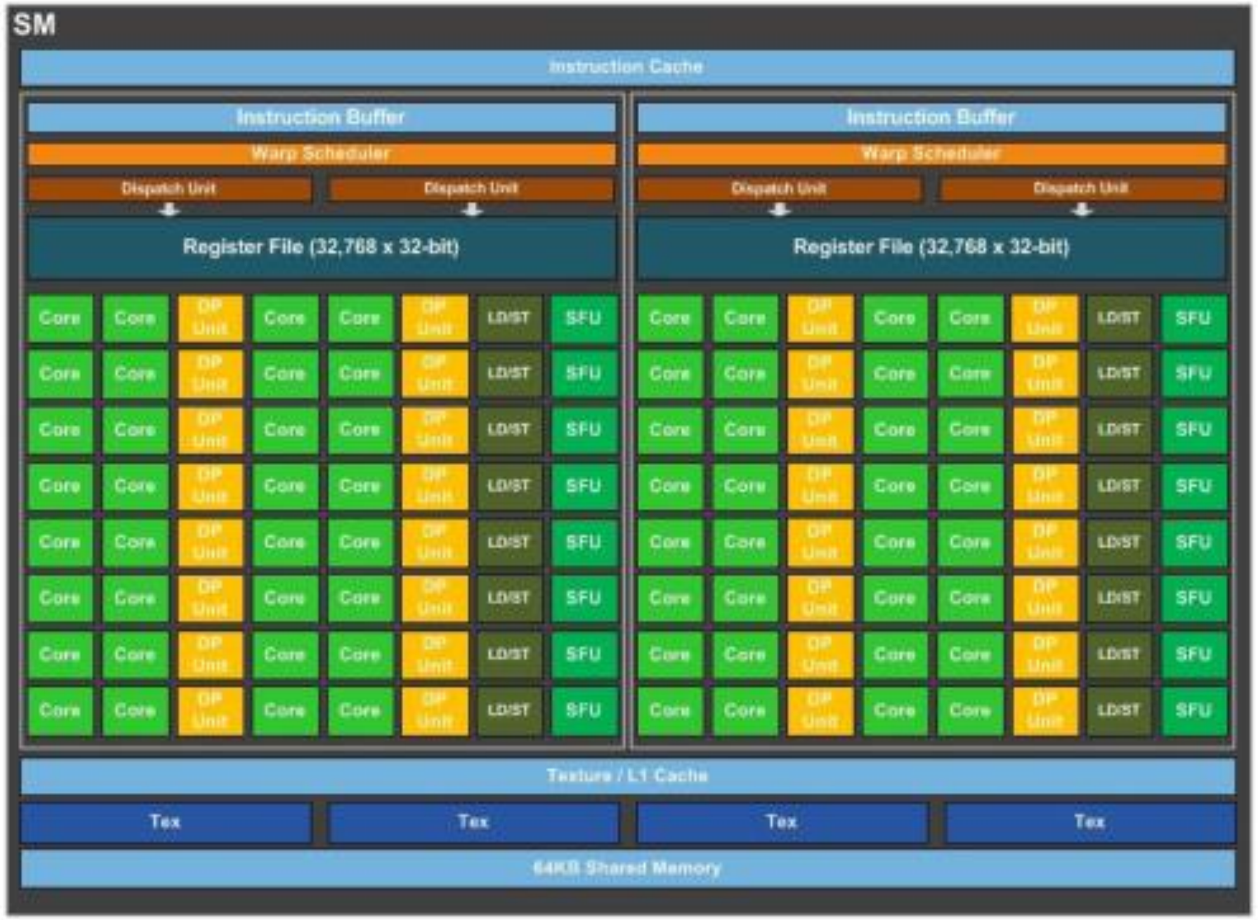


From P100 whitepaper

Streaming multiprocessor (SM): NVIDIA Tesla P100

GP100 SM

	GP100
CUDA Cores	64
Register File	256 KB
Shared Memory	64 KB
Active Threads	2048
Active Blocks	32



Correspondence of Software Component and Hardware

Logical (Software) Component

Hardware Component

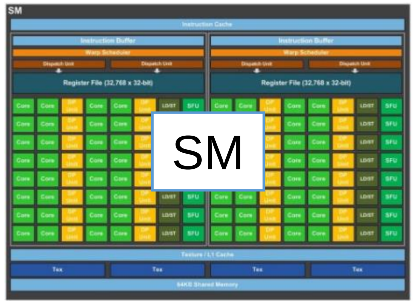
Thread

CUDA core

A thread is treated in a CUDA core.

Block

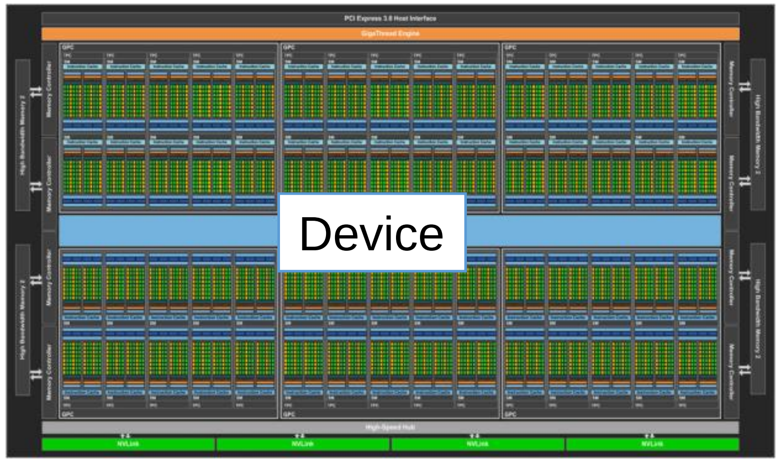
Thread (0,0,0)	Thread (1,0,0)	Thread (2,0,0)
Thread (0,1,0)	Thread (1,1,0)	Thread (2,1,0)



A block is scheduled in a streaming multi-processor. Some blocks correspond to 1 SM.

Grid

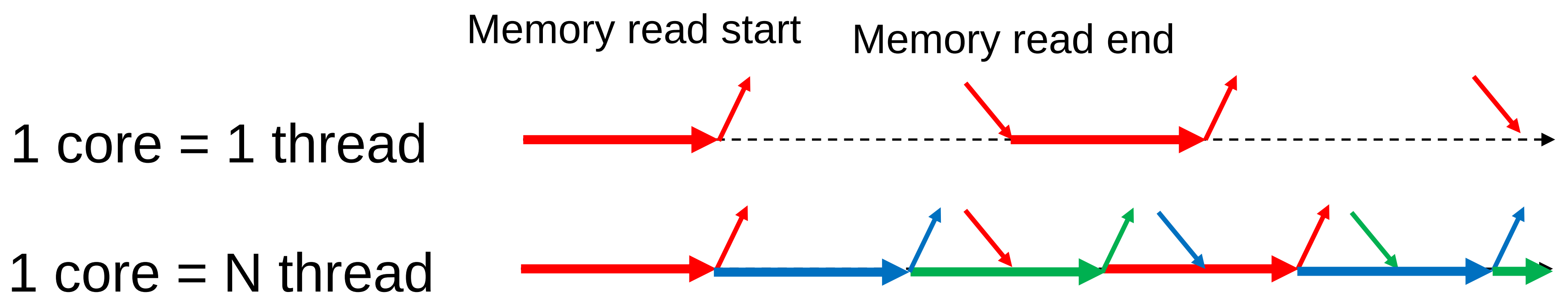
Block (0, 0)	Block (1, 0)	Block (2, 0)
Block (0, 1)	Block (1, 1)	Block (2, 1)



A kernel is booted as a grid.

The Number of Threads and Cores on GPU

- Recommended number of threads
  - ✓ CPU: the number of threads = the number of cores (several 10 threads)
  - ✓ GPU: the number of threads >> the number of cores (several ten thousand~several million threads)
    - Optimal value depends on the balance between other resources.
- Low memory latency by the fast context switch
  - ✓ CPU : Evacuation of register/stack is done by OS (late)
  - ✓ GPU : Overhead is almost zero by hardware support
    - Other thread is executed at free time (stall) due to memory access



Characteristics of Warp Execution

- When a kernel function is executed, a programmer looks like all the threads in the kernel are executed simultaneously. This is logically correct, but it is not on hardware.

  → <u>32 “Threads” are packed into a “Warp” and are processed.</u>

- Warp is a basic execution unit in a SM. When a grid (kernel) is booted, blocks are scheduled and distributed to a SM and the block is divided into warp.
- For example, an 1D block including 128 threads consists of 4 warp as follows.

Warp 0:	Thread 0, Thread 1, Thread 2, ･･･, Thread 31
Warp 1:	Thread 32, Thread 33, Thread 34, ･･･, Thread 63
Warp 2:	Thread 64, Thread 65, Thread 66, ･･･, Thread 95
Warp 3:	Thread 96, Thread 97, Thread 98, ･･･, Thread 127

Warp Divergence

- ➢ 32 threads in a warp are executed simultaneously.

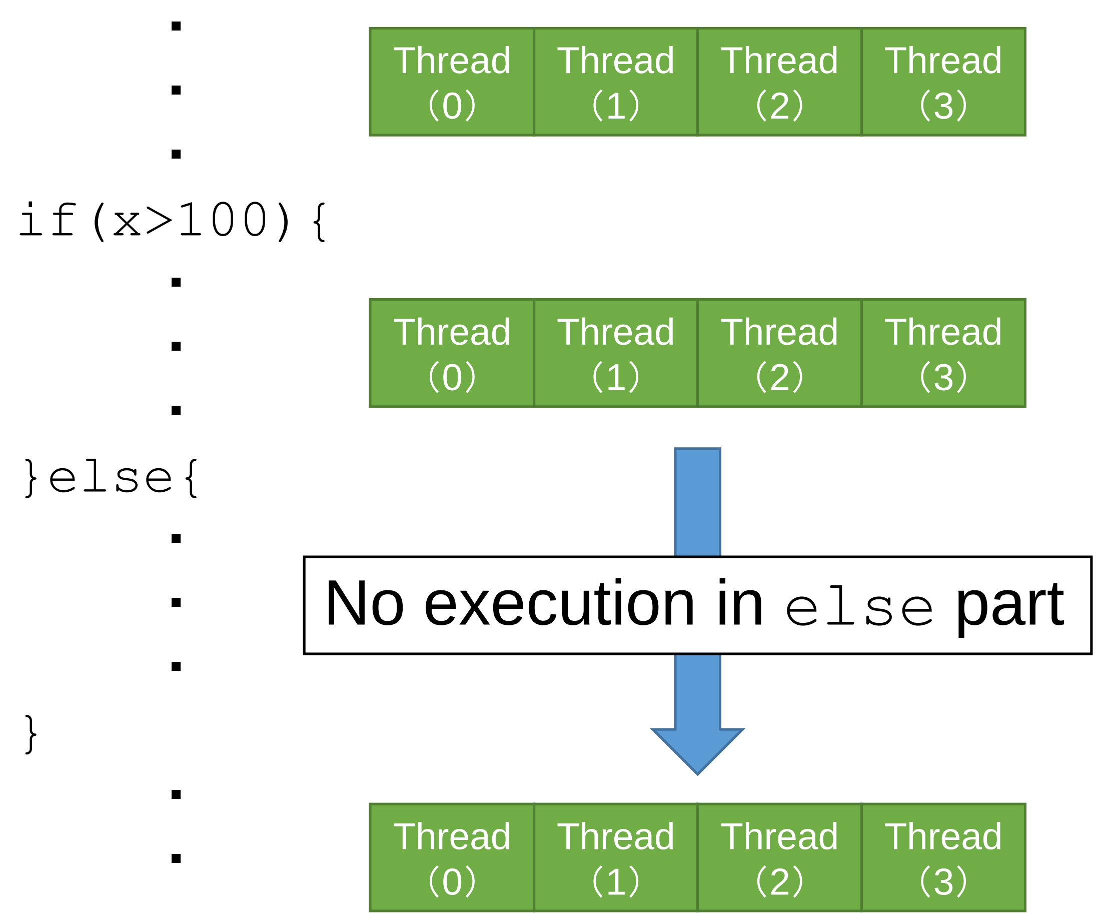
→ What happens if different threads in a warp need to do different things?

- ✓ This is called warp divergence.
- ✓ CUDA will generate correct code to handle this, but to understand the performance you need to understand what CUDA does with it.

if Statements on GPU

(a) Without warp divergence

In the case that total threads are `x>100`

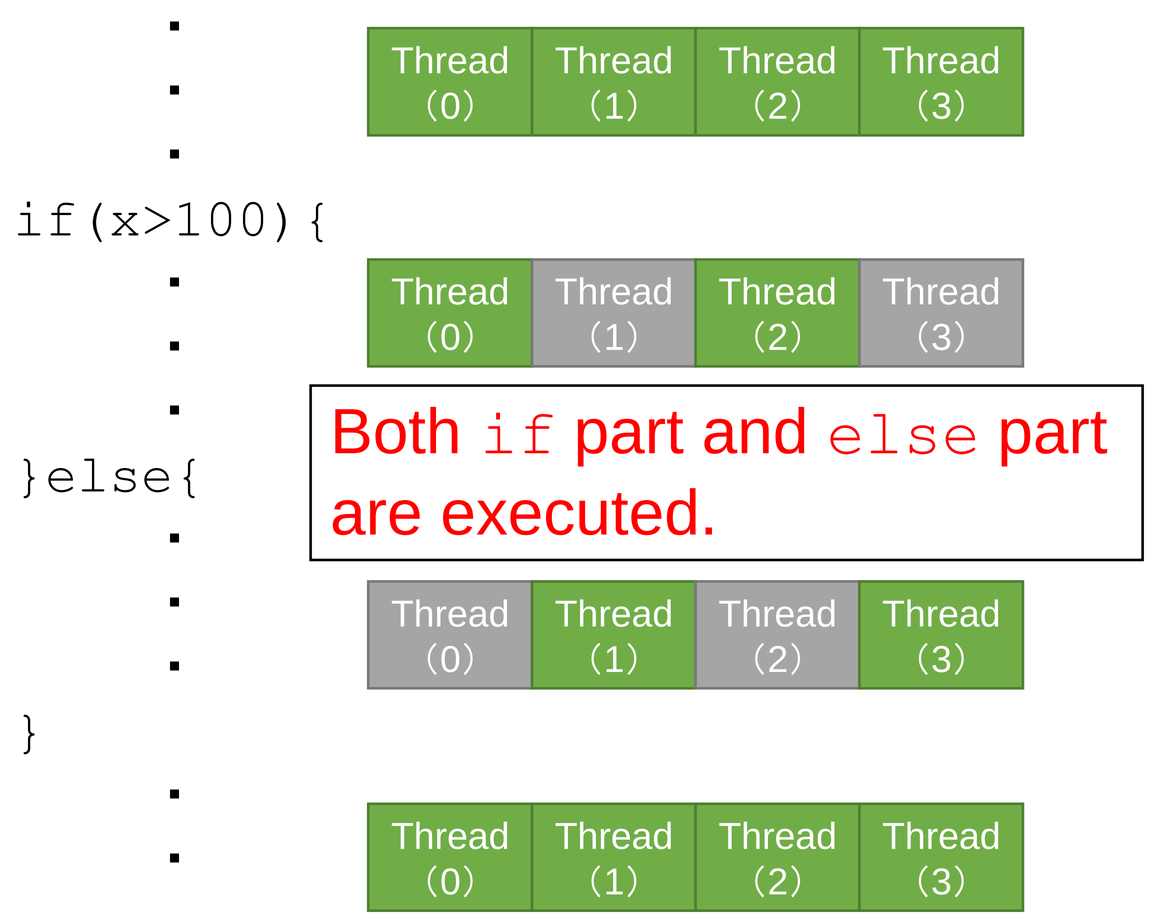


```
    ・
    ・
    ・
if(x>100){
    ・
    ・
    ・
}else{
    ・
    ・
    ・
}
    ・
    ・
```

Thread (0) Thread (1) Thread (2) Thread (3)
Thread (0) Thread (1) Thread (2) Thread (3)
No execution in else part
Thread (0) Thread (1) Thread (2) Thread (3)

(b) With warp divergence

Some threads are `x<100`



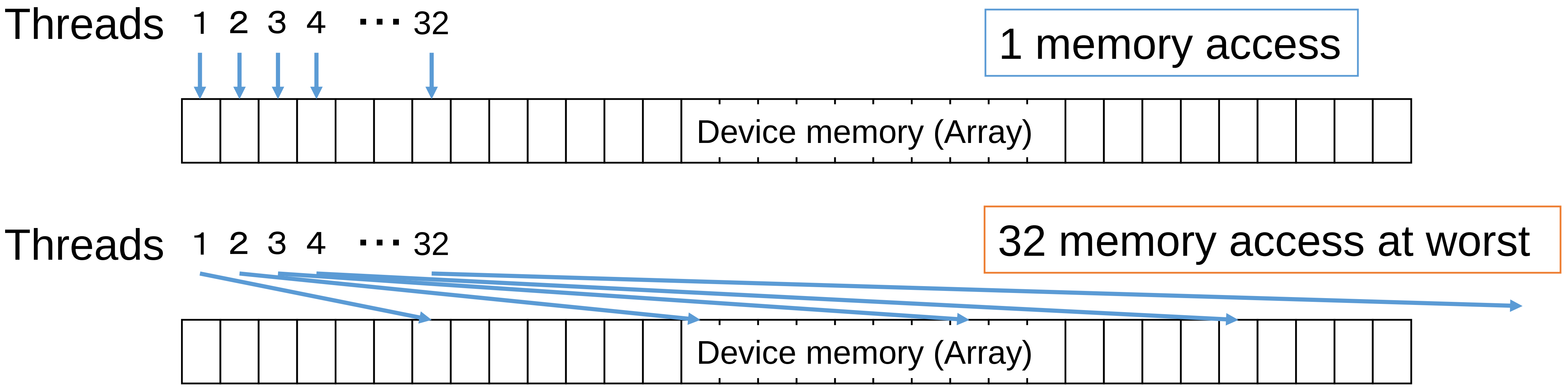
```
    ・
    ・
    ・
if(x>100){
    ・
    ・
    ・
}else{
    ・
    ・
    ・
}
    ・
    ・
```

Thread (0) Thread (1) Thread (2) Thread (3)
Thread (0) Thread (1) Thread (2) Thread (3)
Both if part and else part are executed.
Thread (0) Thread (1) Thread (2) Thread (3)
Thread (0) Thread (1) Thread (2) Thread (3)

Low performance

Coalesced Access and Strided Access

- Threads in a same warp access simultaneously to near memory addresses. This is efficiently for the characteristics of memory. (coalesced access)



The memory access request of a kernel is 128-byte or 32-byte unit. When the access in a warp is within 128-byte, that is only 1 time. If it overflows, the access is repeated for the portion.

Summary: Block and Thread

- ➢ 1 block is executed on 1 SM.
  - ✓ 1 SM shares multiple blocks.
- ➢ 1 thread is executed on 1 CUDA core.
  - ✓ 1 CUDA core shares multiple threads.
  - ✓ Adjacent 32 threads（= warp）are worked simultaneously.
    → When 1 block includes less than 32 threads, some CUDA core are wasted.

Desirable conditions in NVIDIA Tesla P100 in Reedbush

➢ The number of blocks is over 56

and

➢ The number of threads is over 32 (less than 1024)

Time mesurement

```
cudaDeviceSynchronize(); gettimeofday(&t1,NULL);
cudaMemcpy(…, cudaMemcpyHostToDevice);

cudaDeviceSynchronize(); gettimeofday(&t2,NULL);
kernel_func<<<…, …>>>(… );

cudaDeviceSynchronize(); gettimeofday(&t3,NULL);
cudaMemcpy(…, cudaMemcpyDeviceToHost);

cudaDeviceSynchronize(); gettimeofday(&t4,NULL);
```

- This program is an example calling “`gettimeofday`” in C
- `cudaDeviceSynchronize()` is needed.

Command Line Profiler（1/2）

- Simple profile can be obtained by `nvprof` command.

- Write as follows in a job (batch) script
  ```
  nvprof ./a.out
  ```

- After the calculation, simple profile as shown in next page can be written in the standard error (*job_script_file.sh*.`eXXXXXX`).

- The profile doesn’t include the information on CPU (Host) side.

Command Line Profiler（2/2）

```
==26473== NVPROF is profiling process 26473, command: ./a.out
==26473== Profiling application: ./a.out
==26473== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 88.95%  252.44ms         1  252.44ms  252.44ms  252.44ms  matmul_case1(void)
  5.68%  16.118ms         1  16.118ms  16.118ms  16.118ms  matmul_case2(void)
  2.43%  6.9040ms         1  6.9040ms  6.9040ms  6.9040ms  matmul_case3(void)
  1.99%  5.6526ms         6  942.10us  790.89us  1.1248ms  [CUDA memcpy HtoD]
  0.95%  2.7015ms         3  900.49us  802.28us  962.60us  [CUDA memcpy DtoH]

==26473== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 49.67%  508.87ms         6  84.812ms  5.3740us  508.84ms  cudaDeviceSynchronize
 27.82%  285.02ms         9  31.669ms  805.40us  253.41ms  cudaMemcpy
 22.34%  228.88ms         1  228.88ms  228.88ms  228.88ms  cudaDeviceReset
  0.10%  1.0504ms       182  5.7710us     166ns  220.43us  cuDeviceGetAttribute
  0.06%  581.86us         2  290.93us  288.97us  292.89us  cuDeviceTotalMem
  0.01%  92.587us         2  46.293us  43.774us  48.813us  cuDeviceGetName
  0.01%  72.180us         3  24.060us  17.750us  31.588us  cudaLaunch
  0.00%  8.9460us         9     994ns     274ns  5.0750us  cudaGetSymbolAddress
  0.00%  5.0050us         3  1.6680us     533ns  3.6080us  cudaConfigureCall
  0.00%  3.8830us         3  1.2940us     249ns  3.1540us  cuDeviceGetCount
  0.00%  1.9300us         6     321ns     179ns     557ns  cuDeviceGet
```

CUDA Cにおける多次元配列の利用(1/3)

CUDA Cで多次元配列をデバイス側に確保・コピーして計算に用いるのは少しめんどくさい。最も基本的な方法は以下の通り。

1. cudaMallocで領域を確保し、そのポインタを得る。
   例えば、double *dA;
2. cudaMemcpyでホスト側配列AからdAにコピー
3. カーネル関数内でdAの内容にアクセス可能

しかしこの方法では、<u>dAを一次元配列として扱わなければならない</u>。

dA[i][j][k][l]　　←　×このようには使えない
dA[i*max2*max3*max4+j*max3*max4+k*max4+l]
↑　〇動くけど、プログラミングが煩雑になってしまう

＊ちなみにCUDA Fortranでは簡単に多次元配列が利用できます。

CUDA Cにおける多次元配列の利用(2/3)

(方法1)

1. グローバルスコープで変数を定義する際に`__device__`修飾子をつけると、デバイスメモリ上に変数が確保される。
   例:
   ```
   __device__ double dA[max1][max2][max3][max4];
   ```
2. グローバル変数`dA`のアドレスを取得するため`cudaGetSymbolAddress`を呼び出し。この関数は、指定されたデバイスシンボルに関連付けられているグローバルメモリの物理アドレスを取得する。その後、コピー。
   例:
   ```
   double *ptr=NULL;
   cudaGetSymbolAddress((void **)&ptr,dA);
   cudaMemcpy(ptr,A,sizeof(double)*max1*max2*
          max3*max4,cudaMemcpyHostToDevice);
   ```
3. これで、カーネル関数内でも`dA[i][j][k][l]`のように使える。

CUDA Cにおける多次元配列の利用(3/3)

(方法2)

1. グローバルスコープで変数を定義する際に`__device__`修飾子をつけると、デバイスメモリ上に変数が確保される。
   例: `__device__ double dA[max1][max2][max3][max4];`

2. `cudaMemcpyToSymbol`(CPU→GPU)や`cudaMemcpyFromSymbol`(GPU→CPU)を呼び出し。この関数は、グローバルメモリまたはコンスタントメモリで割り当てられた変数へ(または、から)コピー。
   例:
   ```
   cudaMemcpyToSymbol(dA,A,sizeof(double)*max1*max2
           *max3*max4,0);
     cudaMemcpyFromSymbol(A,dA,sizeof(double)*max1*max2
           *max3*max4,0);
   ```

3. これで、カーネル関数内でも`dA[i][j][k][l]`のように使える。

Execution of Sample Programs for CUDA on Reedbush-H (with GPU)

Execution of a CUDA Program (1/2)

1. Move to /lustre directory.
```
$ cd /lustre/gi16/XXXXXX/
```
2. Copy CUDA.tar on /lustre/gi16/c26050 to your own directory.
```
$ cp /lustre/gi16/c26050/CUDA.tar ./
```
3. Extract files from CUDA.tar
```
$ tar xvf CUDA.tar
```
4. Move to CUDA/Hello_CUDA/ directory in Samples.tar.
```
$ cd CUDA/Hello_CUDA/
```
5. Load CUDA environments and PGI compiler
```
$ module load cuda pgi
```

Execution of a CUDA Program (2/2)

6. Compile the source file

for CUDA C : `$ nvcc -gencode arch=compute_60,code=sm_60 hello.cu`

for CUDA Fortran : `$ pgfortran -Mcuda=cc60 hello.cuf`

7. Submit the job

```
$ qsub run.sh
```

8. Confirm the status of submitted job

```
$ rbstat
```

9. After the execution, the following files are generated.

`run.sh.exxxxxx`

`run.sh.oxxxxxx` (`xxxxxx` is Job ID)

10. See the standard output file

```
cat run.sh.oxxxxxx
```

Batch Script for CUDA

```
$ cd /lustre/gi16/XXXXXX
$ cat ./run.sh

#!/bin/sh
#PBS -q h-debug
#PBS -W group_list=gi16
#PBS -l select=1:mpiprocs=1:ompthreads=1
#PBS -l walltime=00:10:00

cd $PBS_O_WORKDIR
. /etc/profile.d/modules.sh

module load cuda pgi

./a.out
```

`h-lecture` when you use GPU

`module` command is needed

Parallelized Hello Program for CUDA C

```
#include<stdio.h>

__global__ void helloFromGPU();

int main(int argc,char *argv[]){
  printf("Hello World from CPU¥n");
  helloFromGPU<<<1,128>>>();
  cudaDeviceReset();
  return 0;
}

__global__ void helloFromGPU(){
  printf("Hello World from GPU thread %d¥n",threadIdx.x);
}
```

Parallelized Hello Program for CUDA Fortran

```
module cudakernel
contains
  attributes(global) subroutine helloFromGPU()
    implicit none
    print *,"Hello World from GPU thread",threadIdx%x
    return
  end subroutine helloFromGPU
end module cudakernel

program main
  use cudafor
  use cudakernel
  implicit none
  integer::ierr
  print *,"Hello World from CPU"
  call helloFromGPU<<<1,128>>>()
  ierr=cudaDeviceReset()
end program main
```

sumarray.cu (CUDA C)

```
#include<stdio.h>
#include<stdlib.h>
#include<cuda.h>
#include<cuda_runtime.h>

__global__ void sum_array(int nemax,double *dA,
                          double *dB,double *dC){
  int ni;
  ni=blockIdx.x*blockDim.x+threadIdx.x;
  dC[ni]=dA[ni]+dB[ni];
  return;
}

int main(int argc,char *argv[]){
  const int nemax=4096,thread=1024;
  int ni;
  double A[nemax],B[nemax],C[nemax];
  double *dA,*dB,*dC;
  // -- set initial value --
  srand(248309);
  for(ni=0;ni<nemax;ni++){
    A[ni]=(double)rand()/RAND_MAX;
    B[ni]=(double)rand()/RAND_MAX;
  }
  // -- allocate arrays on device --
  cudaMalloc((double **)&dA,nemax*sizeof(double));
  cudaMalloc((double **)&dB,nemax*sizeof(double));
  cudaMalloc((double **)&dC,nemax*sizeof(double));
  // -- copy memories from host to device --
  cudaMemcpy(dA,A,nemax*sizeof(double),
             cudaMemcpyHostToDevice);
  cudaMemcpy(dB,B,nemax*sizeof(double),
             cudaMemcpyHostToDevice);
  // -- sum --
  sum_array<<<nemax/thread,thread>>>(nemax,dA,dB,dC);
  // -- copy memories from device to host --
  cudaMemcpy(C,dC,nemax*sizeof(double),
             cudaMemcpyDeviceToHost);
  // -- output --
  for(ni=0;ni<nemax;ni++){
    printf("%d: %lf + %lf = %lf¥n",
           ni,A[ni],B[ni],C[ni]);
  }
  // -- deallocate arrays on Device --
  cudaFree(dA); cudaFree(dB); cudaFree(dC);
  cudaDeviceReset();
  return 0;
}
```

sumarray.cuf（CUDA Fortran）

```fortran
module cudakernel
contains
  attributes(global) subroutine sum_array(nemax,dA,dB,dC)
    implicit none
    integer,value,intent(in)::nemax
    double precision,intent(in)::dA(nemax),dB(nemax)
    double precision,intent(out)::dC(nemax)
    integer::ni
    ni=(blockIdx%x-1)*blockDim%x+threadIdx%x
    dC(ni)=dA(ni)+dB(ni)
    return
  end subroutine sum_array
end module cudakernel

program main
  use cudafor
  use cudakernel
  implicit none
  integer,parameter::nemax=4096,thread=1024
  integer::ni,ierr
  double precision::A(nemax),B(nemax),C(nemax)
  double precision,device,allocatable::dA(:),dB(:),dC(:)
  ! -- set initial value --
  call set_seed(248309)
  do ni=1,nemax
     call random_number(A(ni))
     call random_number(B(ni))
  enddo
  ! -- allocate arrays on device --
  allocate(dA(nemax))
  allocate(dB(nemax))
  allocate(dC(nemax))
  ! -- copy memories from host to device --
  dA=A
  dB=B
  ! -- sum --
  call sum_array<<<nemax/thread,thread>>>(nemax,dA,dB,dC)
  ! -- copy memories from device to host --
  C=dC
  ! -- output --
  do ni=1,nemax
     write(*,'(i5,":",f10.7," +",f10.7," =",f10.7)')&
           ni,A(ni),B(ni),C(ni)
  enddo
  ! -- deallocate arrays on device --
  deallocate(dA); deallocate(dB); deallocate(dC)
  ierr=cudaDeviceReset()
end program main
```

Practice 1: Hello and sumarray program

- Confirm and execute the source file “`hello.cu`” or “`hello.cuf`” and change the number of threads.

- Move to sumarray/ directory.
- Confirm and execute the source file “sumarray.cu” or “sumarray.cuf”.
- Check the execution time by changing the number of threads. (Obviously, the execution time may decrease with increasing the number of threads)

Practice 2: Matrix-Matrix Multiplication

- Move to `Matmul_CUDA/` directory
- Confirm how calculation is executed by reading the source file “`matmul.cu`” or “`matmul.cuf`”.
- Compare the execution time of CPU ver., OpenMP ver. and 3 cases of CUDA ver. in “`matmul.cu`” or “`matmul.cuf`” (the execution time includes data transfer time). What is difference in case 1～3 of CUDA ver?
- Execute by changing the matrix size, the number of blocks and threads

Practice 3: use of shared memory

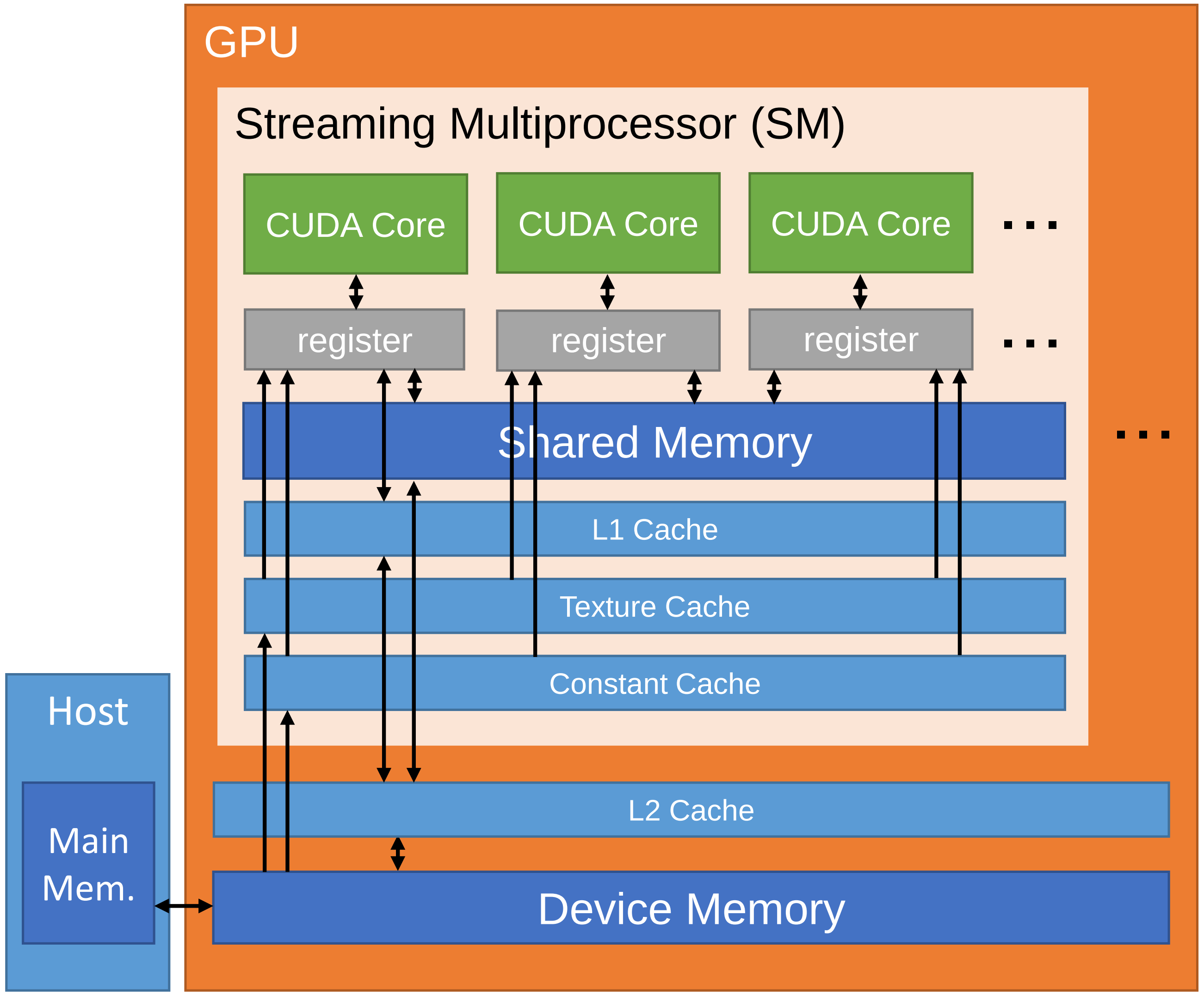
- Move to Mattrans/ directory
- Confirm how calculation is executed by reading the source file “`mattrans.cu`” or “`mattrans.cuf`” and how to use shared memory.
- Compare the execution time of the follows: 1. naive copy of matrix, 2. copy of matrix using shared memory, 3. naive copy to transpose matrix, 4. copy to transpose matrix using shared memory, 5. copy to transpose matrix using shared memory without bank conflict by padding

CUDA Architecture（Physical Configulation）

In GPU, not only thread but also memory has hierarchical structure.

- Register（near CUDA Core）
- Shared memory（is shared by each CUDA core in SM）
- L1 cache（ is shared by each CUDA core in SM ）
- Texture cache（Read only）
- Constant cache（Read only）
- L2 cache（is shared in each SM）
- Device memory

These configuration depends slightly on the generation of GPU.



Copy to Transpose Matrix（C）

$$A = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix} \rightarrow A^T = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

naive imprementation

```
__device__ double idata[nimax][njmax],odata[nimax][njmax];

__global__ void transposeNaive(){
  int ni,nj;
  ni=blockIdx.y*blockDim.y+threadIdx.y;
  nj=blockIdx.x*blockDim.x+threadIdx.x;
  odata[nj][ni]=idata[ni][nj];
  return;
}
```

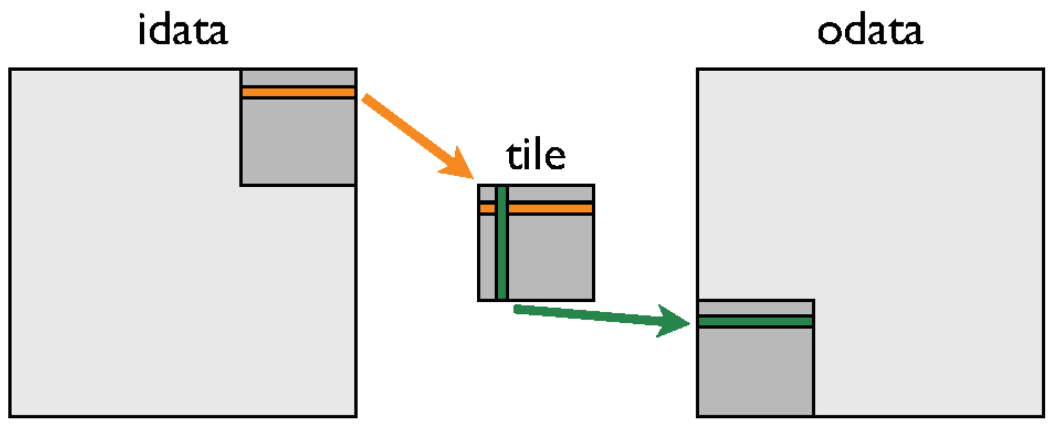
Copy to Transpose Matrix（Fortran）

$$A = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix} \rightarrow A^T = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

naive imprementation

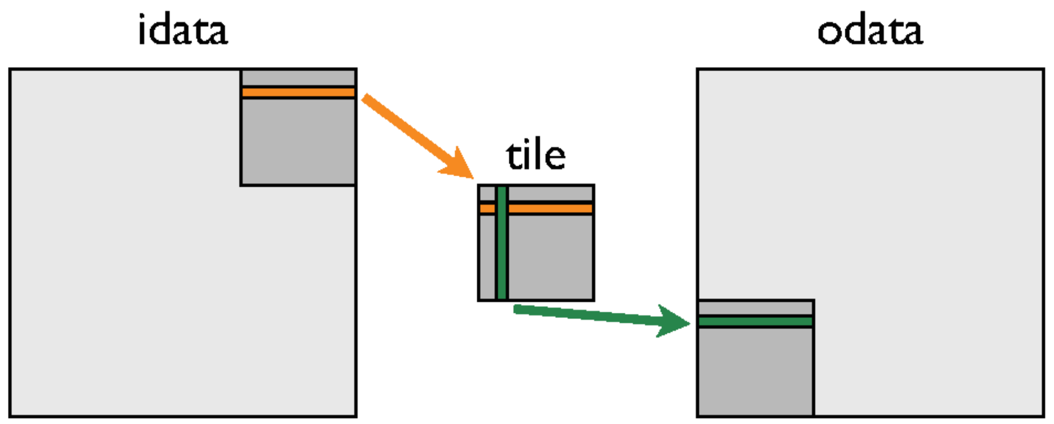
```
attributes(global) subroutine transposeNaive(idata,odata)
    double precision,intent(in)::idata(nimax,njmax)
    double precision,intent(out)::odata(nimax,njmax)
    integer::ni,nj
    ni=(blockIdx%x-1)*blockDim%x+threadIdx%x
    nj=(blockIdx%y-1)*blockDim%y+threadIdx%y
    odata(nj,ni)=idata(ni,nj)
    return
  end subroutine transposeNaive
```

Copy to Transpose Matrix Using Shared Memory (C)



```
__global__ void transposeShared(){
  int ni,nj;
  __shared__ double tile[BS][BS];
  ni=blockIdx.y*blockDim.y+threadIdx.y;
  nj=blockIdx.x*blockDim.x+threadIdx.x;
  tile[threadIdx.y][threadIdx.x]=idata[ni][nj];
  __syncthreads();
  ni=blockIdx.x*blockDim.x+threadIdx.y;
  nj=blockIdx.y*blockDim.y+threadIdx.x;
  odata[ni][nj]=tile[threadIdx.x][threadIdx.y];
  return;
}
```

Copy to Transpose Matrix Using Shared Memory (Fortran)



```
attributes(global) subroutine transposeShared(idata,odata)
    double precision,intent(in)::idata(nimax,njmax)
    double precision,intent(out)::odata(nimax,njmax)
    double precision,shared::tile(BS,BS)
    integer::ni,nj
    ni=(blockIdx%x-1)*blockDim%x+threadIdx%x
    nj=(blockIdx%y-1)*blockDim%y+threadIdx%y
    tile(threadIdx%x,threadIdx%y)=idata(ni,nj)
    call syncthreads()
    ni=(blockIdx%y-1)*blockDim%y+threadIdx%x
    nj=(blockIdx%x-1)*blockDim%x+threadIdx%y
    odata(ni,nj)=tile(threadIdx%y,threadIdx%x)
    return
  end subroutine transposeShared
```