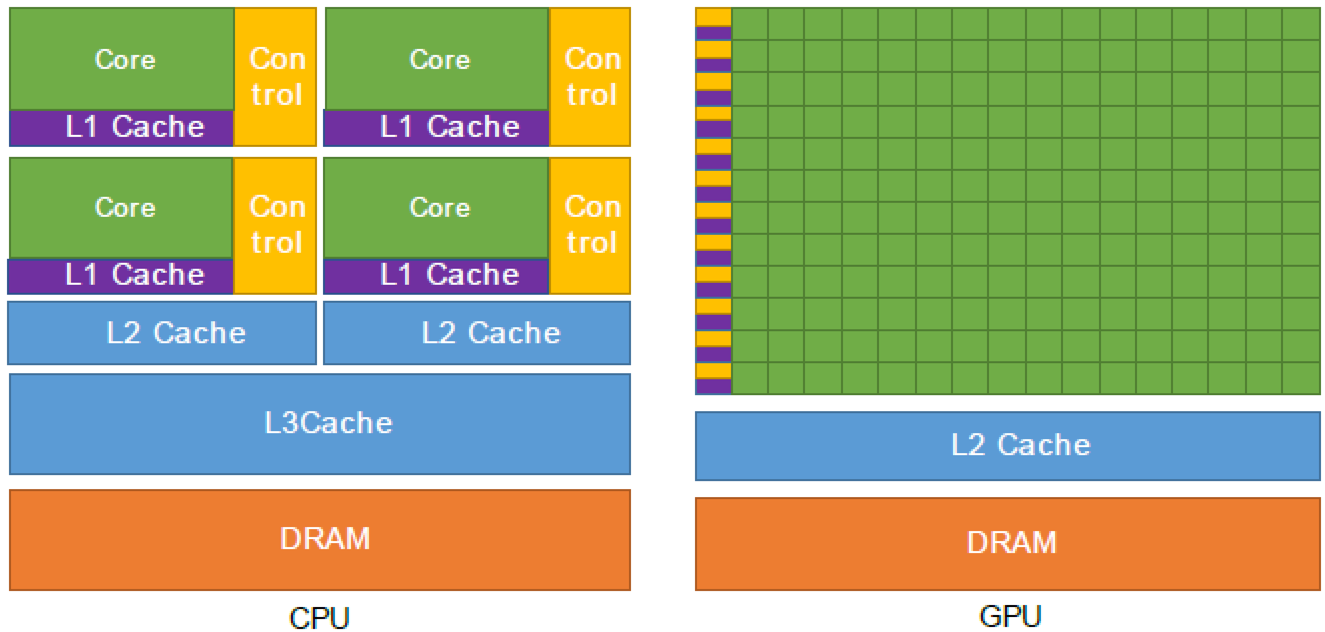
Parallel Scientific Computation

CUDA 1

J.-H. Parq
IPCST
Seoul National University

CPU vs GPU

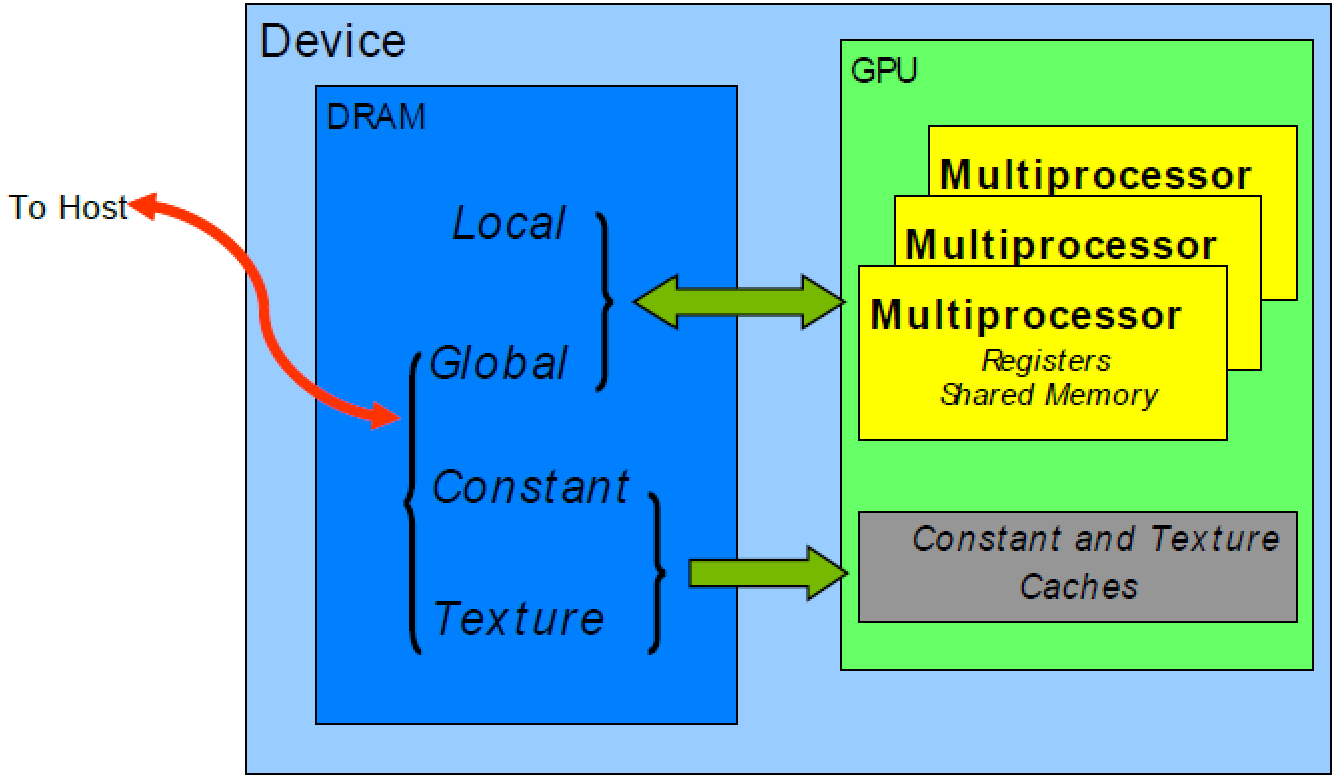


- GPU - many cores, small cache
  - Figure from NVIDIA CUDA C Programming Guide

What is GPGPU?

- General-Purpose computing on Graphics Processing Units
- Using GPUs not for graphics but for mathematical computation
- Transferring data to GPUs and computing
- Stream processing
  - Cf.) pipelining
- Interfaces: CUDA, OpenCL, OpenACC, ……

Memory Spaces on a GPGPU Device



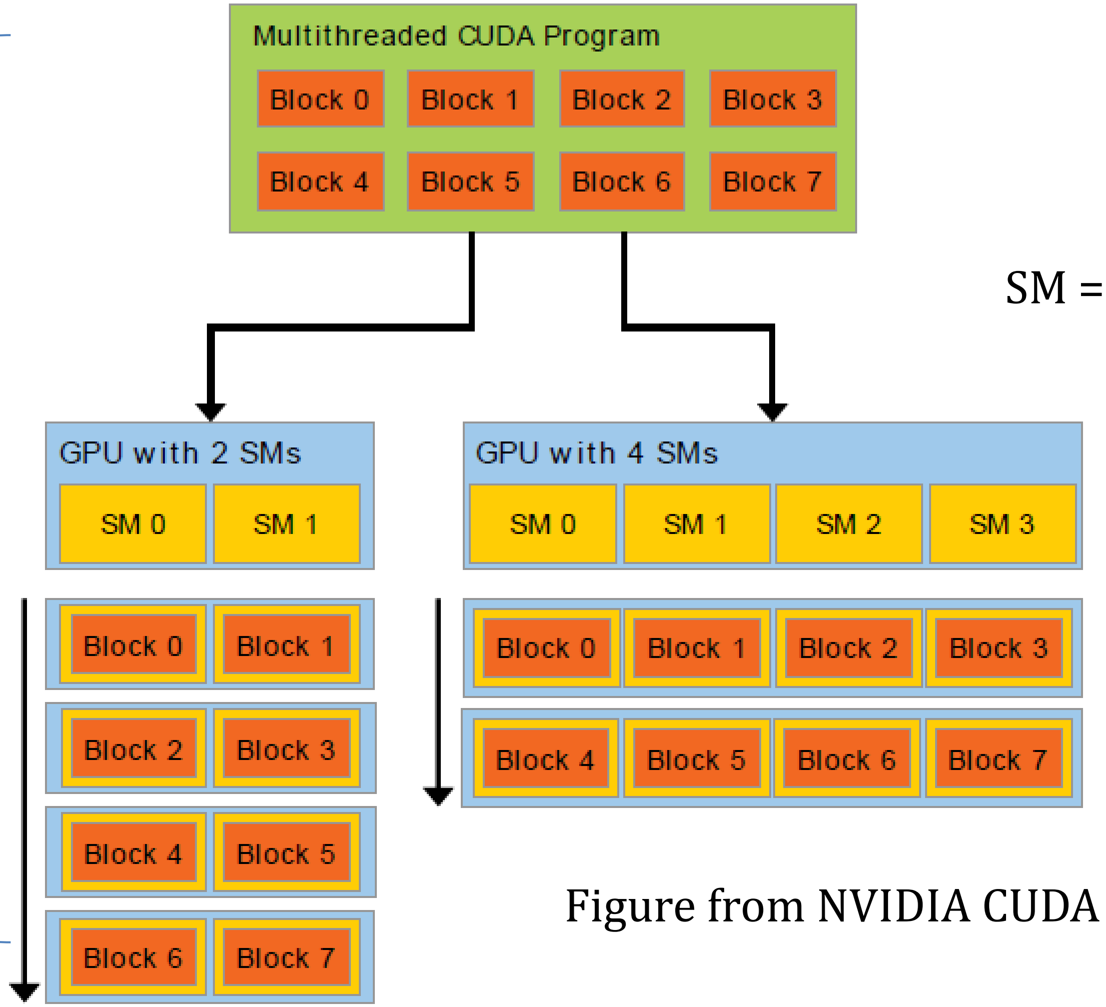
- Detailed structure depends on the GPU compute capability.

Figure from NVIDIA CUDA C Best Practices Guide

What is CUDA?

- Compute Unified Device Architecture
- General purpose parallel computing platform and programming model by NVIDIA
  - Only for Nvidia GPUs
- It provides a small set of extensions to C, C++, or Fortran.
    - Wrappers are available for Python, MATLAB, R, …
- It supports heterogeneous computing using both CPUs and GPUs.
- SIMT (Single Instruction Multiple Threads)
  - Scalable programming model

Scalable Programming



SM = Streaming Multiprocessor

Figure from NVIDIA CUDA C Programming Guide

CUDA Programs

- CUDA C
  - Extension: .cu
  - Compiling
    - nvcc xx.cu
    - ➢ See 'CUDA Compiler Driver NVCC' for options
- CUDA Fortran
  - Extension: .cuf, .CUF (+preprocessor)
  - Based on F90 or Fortran 2003
  - Compiling
    - NVIDIA HPC compiler: nvfortran xx.cuf
    - Portland group compiler: pgf90 xx.cuf / pgfortran xx.cuf

Data Transfer

- Between the host CPU and the device GPU
- In C,
  - cudaMemcpy(dev_ptr, host_ptr, size, cudaMemcpyHostToDevice)
  - cudaMemcpy(host_ptr, dev_ptr, size, cudaMemcpyDeviceToHost)
- In Fortran
  - Assignment statements
  - You can also use cudaMemcpy in the runtime routines.

Data Transfer Simple Example

CUDA Fortran

```
......
Use cudafor
Integer , parameter :: N = 1024
Real :: H_a(N), H_b(N)
Real, device :: D_a(N)
......
D_a = H_a
......
H_b = D_a
......
```

CUDA C

```
......
 size_t size = N * sizeof(float);
 float *h_a = (float*)malloc(size);
 float *d_a; cudaMalloc(&d_a, size);
......
 cudaMemcpy(d_a, h_a, size,
    cudaMemcpyHostToDevice);
......
 cudaMemcpy(h_a, d_a, size,
    cudaMemcpyDeviceToHost);
 cudaFree(d_a);
......
```

Kernels

- User-defined subroutines executing on GPUs

- CUDA Fortran

```
Attributes(global) subroutine k(a,b)
   ......
End subroutine

Program ......
   ......
   Call k<<<M, N>>>( a, b )
   ......
End Program ......
```

- CUDA C

```
__global__ void k(float *a, float *b){
   ......
}

int main() {
   ......
   k<<<M, N>>>(a, b);
   ......
}
```

Thread Hierarchy

- Threads in one block
  - An example in CUDA C Programming Guide

```
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
int main()
{
    ...
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Total threads in one block (= blockDim.x)

# of blocks = 1

```
int i = blockIdx.x;
```

```
VecAdd<<<M, 1>>>(A, B, C);
```

Fortran

```
Integer, value :: I
I = threadIdx%x
I = blockIdx%x
```

blockDim%x Fortran

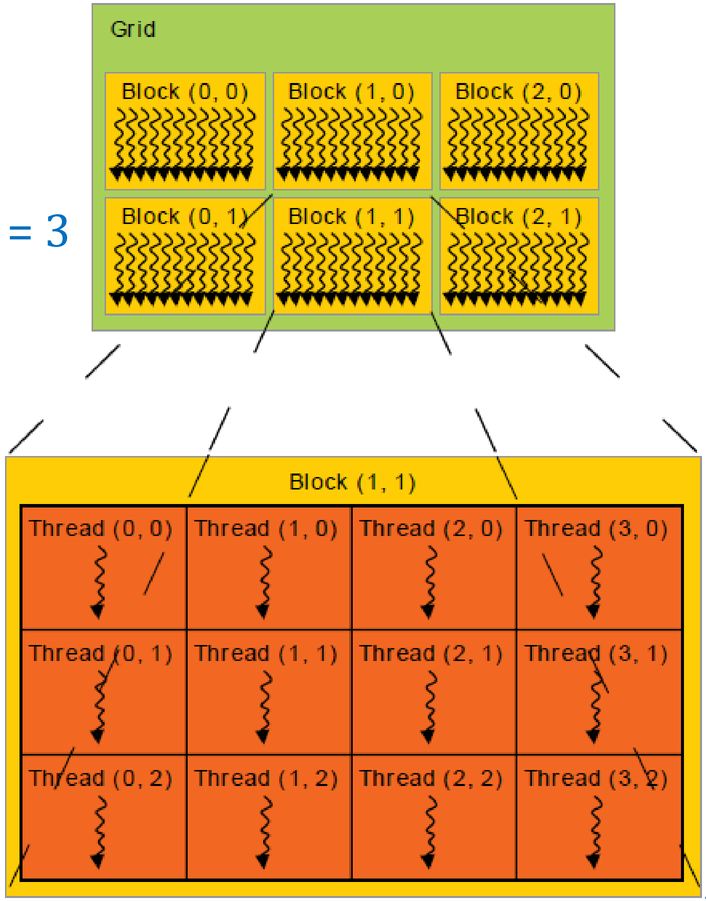
Fortran

```
Call VecAdd<<<1, M>>>(A, B, C)
Call VecAdd<<<M, 1>>>(A, B, C)
```

Thread Hierarchy

- Grid of thread blocks
  - 2D example (CUDA C)
    - threadIdx.y = 0 ~ 2
    - blockIdx.y = 0 ~ 1
  - 3D is also possible.

blockDim.y = 3




```
main()
{
    ...
    dim3 TPB(4,3);
    dim3 nBlocks(3,2);
    kernel<<<nBlocks, TPB>>>(A, B, C);
    ...
}
```

Usually (16,16)
***i.e.*, 256 threads**
per block

Figure from NVIDIA CUDA C Programming Guide

Thread Hierarchy

- Grid of thread blocks
  - 2D example (CUF)
    - threadIdx%y = 0 ~ 2
    - blockIdx%y = 0 ~ 1

blockDim%y = 3

```
...
USE CUDAFOR
...
type (dim3) :: TPB, GRID
TPB = dim3(4,3,1)
GRID = dim3(3,2,1)
Call kernel<<<GRID, TPB>>>(A, B, C)
...
```

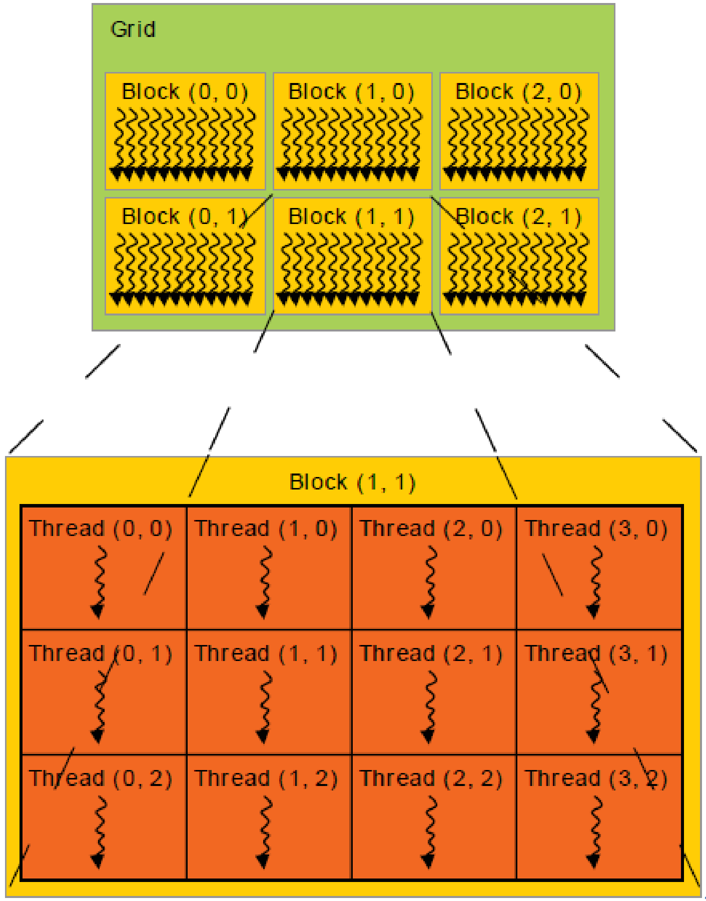


Figure from NVIDIA CUDA C Programming Guide

Memory Hierarchy

- Multiple memory spaces
- Speed

  Global < Local < Shared < Regs.
- Capacity

  Regs. < Shared < Local < Global

  ❖Registers ~ hidden local memory

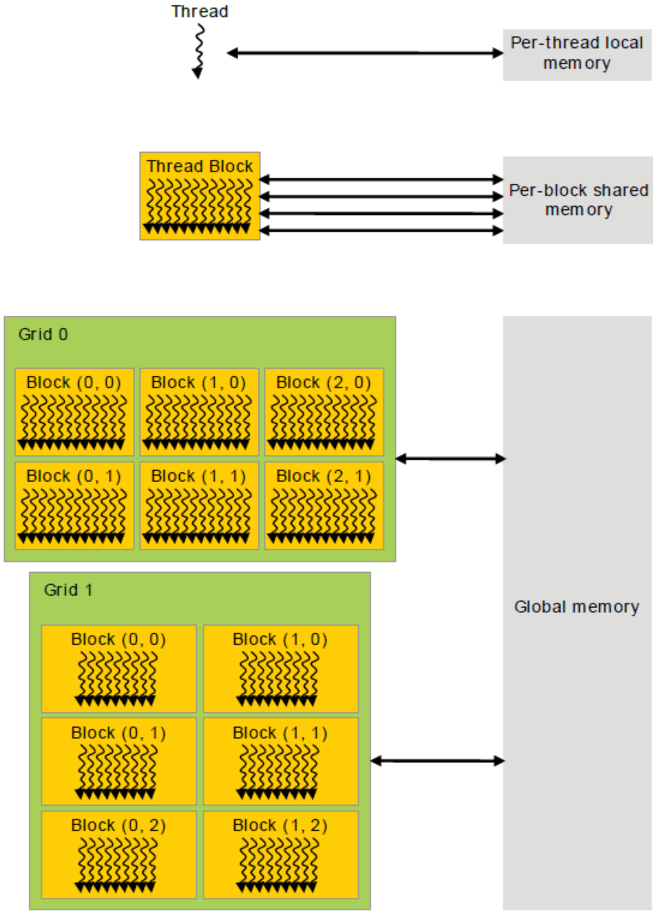


Figure from NVIDIA CUDA C Programming Guide

General CUDA Program Structure

1. **Initialization**
   – Usually implicit in the program
   – Selecting a GPU to run on
2. **Global memory allocation on the GPU**
   – Maybe implicit or not
3. **Data transfer: Host (usually CPU) → GPU**
4. **Launch GPU kernels from the host**
5. **Setting variables & GPU computation**
   – Variables: Local on a thread or Shared by a block
6. **Result transfer: GPU → Host (usually CPU)**
7. **Output & memory freeing on the GPU**
   – Freeing may be implicit or not

Matrix Transpose (CUDA C)

```
#include <stdio.h>
#define nTx 16          //   This should be 16 x n
#define nTy 16
main()
{
......
  cudaMalloc(&inmat_d, size);
  cudaMalloc(&outmat_d, size);
  nBx = Nrow/nTx + (Nrow%nTx != 0);
  nBy = Ncol/nTy + (Ncol%nTy != 0);
  dim3 grid(nBx, nBy), threads(nTx, nTy);
  cudaMemcpy(inmat_d, inmat_h, size, cudaMemcpyHostToDevice);
```

Matrix Transpose (CUDA C)

```
 transpose<<<grid, threads>>>(inmat_d, outmat_d, Nrow, Ncol);
 cudaMemcpy(outmat_h, outmat_d, size, cudaMemcpyDeviceToHost);
……
}

__global__ void transpose(double *inmat_d, double *outmat_d, int Nrow,
    int Ncol)
{
 int i = blockIdx.x * blockDim.x + threadIdx.x;
 int j = blockIdx.y * blockDim.y + threadIdx.y;

 outmat_d[j*Nrow + i] = inmat_d[i*Ncol + j];
}
```

Matrix Transpose (CUF)

```
program MT
    use cudafor
    use MTkernel
    ......
    type(dim3) :: grid, threads
    ......
    grid = dim3(ceiling(real(nrow)/ntx), ceiling(real(ncol)/nty), 1)
    threads = dim3(ntx,nty,1)
    inmat_d = inmat_h
    call transpose<<<grid, threads>>>(inmat_d, outmat_d, nrow, ncol)
    outmat_h = outmat_d
    ......
end program MT
```

Matrix Transpose (CUF)

```
Module MTkernel
    attributes(global) subroutine transpose(inmat_d, outmat_d, nrow,  &
        ncol)
        real(8), intent(in) :: inmat_d(nrow, ncol)
        real(8), intent(out) :: outmat_d(ncol, nrow)
        integer :: i, j
        i = (blockIdx%x-1) * blockDim%x + threadIdx%x
        j = (blockIdx%y-1) * blockDim%y + threadIdx%y
        outmat_d(j, i) = inmat_d(i, j)
    end subroutine transpose
End Module MTkernel
```

Coalesced Memory Access

- All threads in a block or of a warp should access global memory in a range, no same address, fitting in a cache line.
  - Warp: a group of 32 threads in SIMT architecture
- The coalescing condition depends on the compute capability of the GPU.
- Good access
  - Ex.) 4B words

    Coalesced access

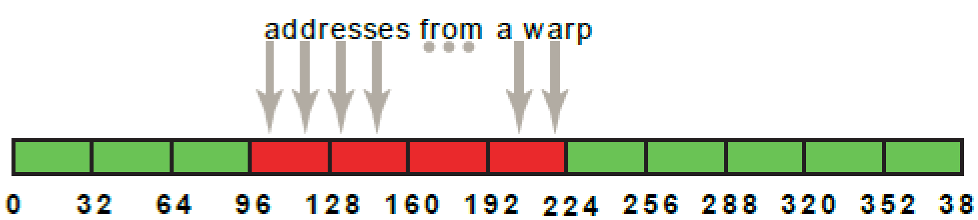
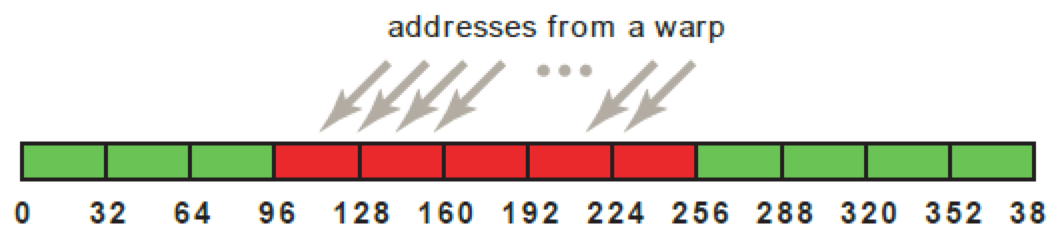


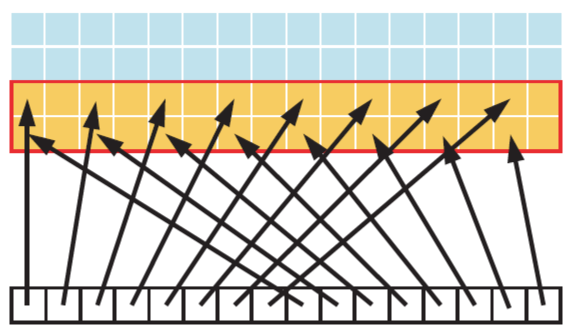
Figure from NVIDIA CUDA C Best Practices Guide

Coalesced Memory Access

- A little bit bad access
  - Ex.) misaligned sequential addresses within five 32B segments



- Bad access



- Shared memories can increase access efficiency.

Figures from NVIDIA CUDA C Best Practices Guide

Matrix Transpose (CUDA C)

```
__global__ void transpose(double *inmat_d, double *outmat_d, int Nrow,
    int Ncol)
{
  __shared__ double tile[nTx][nTy]
  int bx = blockIdx.x * blockDim.x ;
  int by = blockIdx.y * blockDim.y ;
  int i = threadIdx.x;
  int j = threadIdx.y;

  tile[i][j] = inmat_d[(bx+i)*Ncol + by+j];
  __syncthreads();
  outmat_d[(by+i)*Nrow + bx+j] = tile[j][i];
}
```

Matrix Transpose (CUF)

```
Module Mtkernel
    Integer, parameter :: ntx = 16
    Integer, parameter :: nty = 16

Contains
    Attributes(global) Subroutine transpose(inmat_d, outmat_d, &
            nrow, ncol)
            Real(8), intent(in) :: inmat_d(nrow, ncol)
            Real(8), intent(out) :: outmat_d(ncol, nrow)
            Real(8), shared :: tile(ntx, nty)
            Integer :: i, j
```

Matrix Transpose (CUF)

```
        i = (blockIdx%x-1) * blockDim%x + threadIdx%x
        j = (blockIdx%y-1) * blockDim%y + threadIdx%y
        tile(threadIdx%x, threadIdx%y) = inmat_d(i, j)

        Call syncthreads()

        i = (blockIdx%y-1) * blockDim%y + threadIdx%x
        j = (blockIdx%x-1) * blockDim%x + threadIdx%y
        outmat_d(i, j) = tile(threadIdx%y, threadIdx%x)
    End Subroutine transpose
End Module MTkernel
```

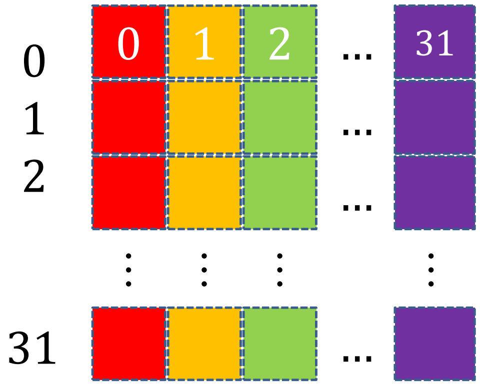
Bank Conflicts

- **Bank**: equally-sized memory units in shared memory. Simultaneously accessed.
  - 32 banks for compute capability 3.x or higher
- Bank rules
  - 1 thread 1 bank access → OK
  - 2 thread 1 bank & the same word → OK
    - Broadcast reading
  - 2 thread 1 bank & different words → Conflict!
  - ❖ Word: 32-bit mode or 64-bit mode

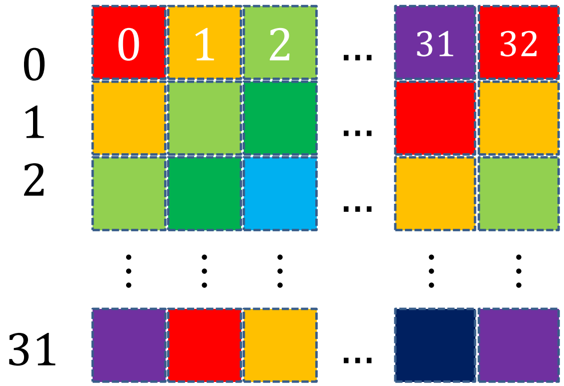
Bank Conflicts

- 2-D array example
  - 32 x 32 array (C)
    - Red: bank 0
    - Orange: bank 1

      ⋮



- Remedy: array padding
  - __shared__ double tile[32][33]
  - Real(8), shared :: tile(33, 32)



Kernel Do Directive in CUF

- Similar to the OpenMP do directive
  - See more cases in the references.

- Ex. 1)

```
!$cuf kernel do(2) <<< (*,*), (32, 4)>>>
Do j = 1, m
   Do i = 1, n
      a(i,j) = b(i,j) + c(i,j)
   End do
End do
```

- (2): collapse
- (32,4): 32 thread ID x → inner loop
  4 thread ID y → outer loop

- Ex. 2)

```
sum = 0.0
!$cuf kernel do <<<*,*>>>
Do i = 1, nx
   sum = sum + d_a(i)
End do
```

- Automatic reduction
- *,* : set by the compiler

※Make sure to check restrictions.

OpenACC

- Acronym for **open acc**elerators
- API to provide easy parallel programming for accelerators
  - Similar to OpenMP
  - Not only for Nvidia GPUs but also for AMD GPUs
- Supported languages: Fortran, C, C++
- Compilers
  - Commercial / semi-commercial (NVIDIA) / academic (from universities) / free (GCC)

OpenACC

- Directives
  - Similar to OpenMP
    - #pragma acc …… or !$acc ……
- Pallelizing loops
  - ‘parallel loop’ directive: similar to ‘parallel do/for’
  - ‘kernels’ directive: automatic optimization
    - Depending on your compiler
  - ‘loop’ directive: direction for the following loop
    - Used in ‘parallel’ or ‘kernels’ blocks
    - Clauses: private, reduction, collapse, ……
  - ‘kernels loop’ directive

OpenACC Examples

Fortran

```fortran
!acc kernels
Do i=1, N
  x(i) = i
  y(i) = 0
End do
Do i=1, N
  y(i) = x(i)*3.0 + y(i)
End do
!acc end kernels
```

C

```c
#pragma acc kernels
{
  for (i=0, i<N, i++) {
    x[i] = (float)i;
    y[i] = 0.0f;
  }
  for (i=0, i<N, i++)
    y[i] += 3.0f*x[i];
}
```

OpenACC Examples

Fortran

```
!$acc parallel loop
do j=1,m-2
  !$acc loop
  do i=1,n-2
    A(i,j) = Anew(i,j)
  end do
end do
```

C

```
#pragma acc parallel loop
 for ( int j = 1; j < n-1; j++)
{
  #pragma acc loop
  for ( int i = 1; i < m-1; i++ )
  {
    A[j][i] = Anew[j][i];
  }
}
```

OpenACC

- Implicit data transfer at the beginning and the end of ‘kernels’ or ‘parallel’ blocks → **overhead**
- Explicit data transfer
  - ‘data’ directive: defining a data block
    - Controlling memory allocation on a device at the block head & Data transfer depending on clauses
    - Clauses: copy(=copyin+copyout), copyin(host→device), copyout(host→device), create, present, deviceptr
    - **No data transfer** for the variables at the front and the end of ‘kernels’ or ‘parallel’ blocks **in the data block**
  - ‘update’ directive: data synchronization (no block)
    - Clauses: device(host→device), self(device→host)

OpenACC

- Example of ‘data’ directive (C)

```
#pragma acc data copy(A), create(Anew)
{
 while ( err > tol && iter < iter_max ) {
  err=0.0;
  #pragma acc parallel loop reduction(max:err)
  for( int j = 1; j < n-1; j++) {
   for(int i = 1; i < m-1; i++) {
    Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] + A[j-1][i] + A[j+1][i]);
    err = max(err, abs(Anew[j][i] - A[j][i]);
   }
  }
……
```

OpenACC

- Another example of ‘data’ directive (Fortran)
  - Clause: p… = present or …
    - Ex.: pcopy = present or copy

```
!acc data pcopyout(x(1:M)), pcreate(y(1:M))
!acc parallel loop
Do i = 1, N
  x(i) = i
  y(i) = 0
End do
......
!acc parallel loop
Do i = 1, N
  y(i) = y(i) + x(i)
End do
!acc end data
```

References

- CUDA C Programming Guide
  - CUDA C++ Programming Guide
- CUDA C Best Practices Guide
  - CUDA C++ Best Practices Guide
- CUDA Fortran Programming Guide
- Wikipedia

References

- Introduction to CUDA Fortran
  - GPU Technology Conference 2013

- OpenACC homepage

https://www.openacc.org/

- OpenACC getting started guide

- J. Sanders & E. Kandrot, CUDA by Example