Cineca
TRAINING
High Performance
Computing 2017

Programming techniques for heterogeneous architectures

Pietro Bonfa' – p.bonfa@cineca.it

SuperComputing Applications and Innovation Department

CINECA

Heterogeneous computing

Gain performance or energy efficiency not just by adding the same type of processors, but **by adding dissimilar coprocessors**, usually incorporating **specialized** processing capabilities to handle **particular tasks**.

WIKIPEDIA
The Free Encyclopedia

DISCLAIMER NOTE, some slides and images are courtesy of: NVIDIA (CUDA manual, CUDA Fortran examples), CUDA Team in CINECA (Luca Ferraro, Sergio Orlandini, Stefano Tagliaventi) and/or specified sources.

My aim for today

Overview of different approaches for heterogeneous computing…

TRY BEFORE YOU BUY

...discuss their strengths and weaknesses...

… and some points to keep in mind!

Outline

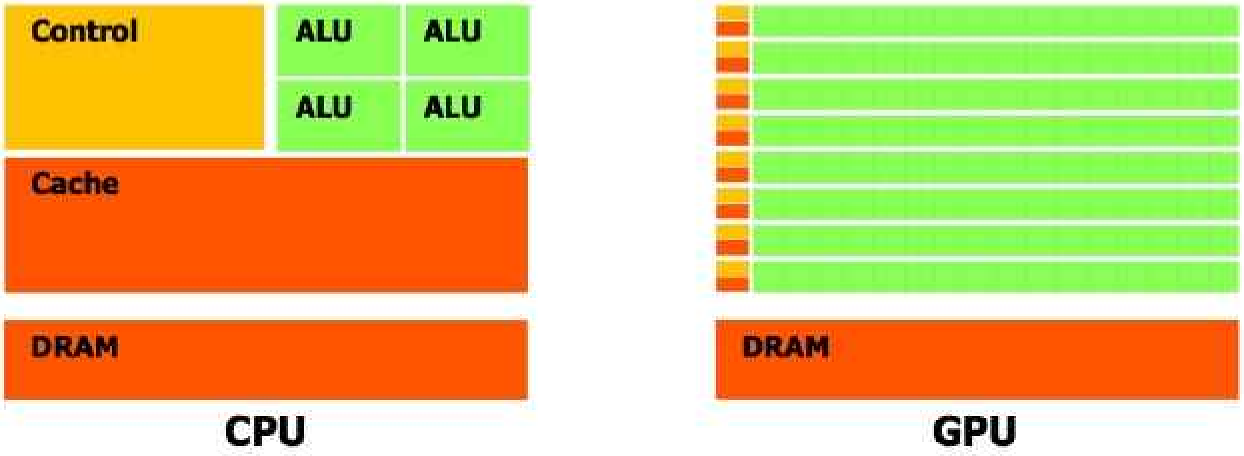
- Introduction to accelerators
- GPU Architecture
- Programming models
- Some recommendations

GPUs

General **P**urpose **G**raphical **P**rocessing **U**nits

GPU Architecture

- Difference between GPU and CPU

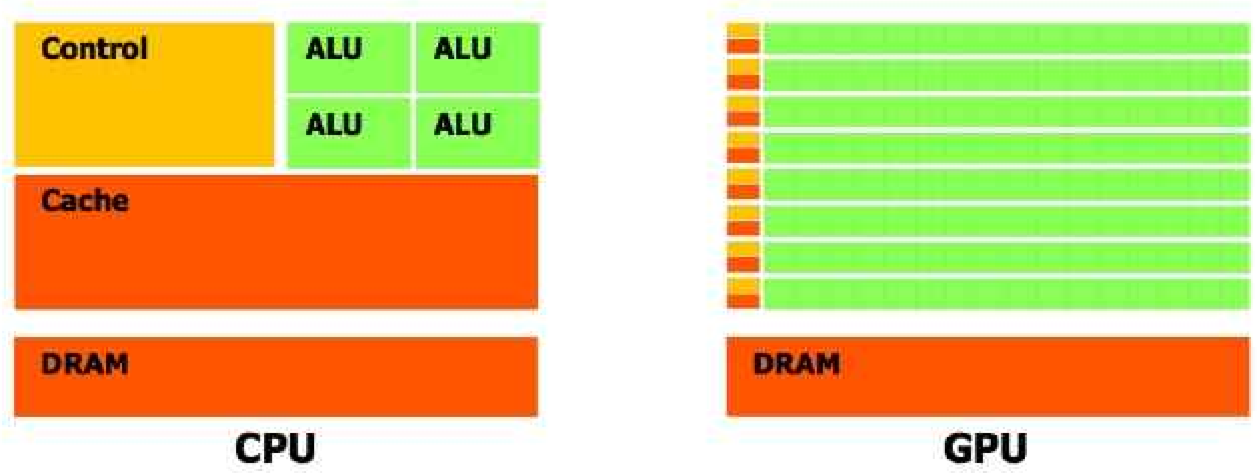


More transistors devoted to data processing

(but less optimized memory access and speculative execution)

GPU Architecture

- Difference between GPU and CPU

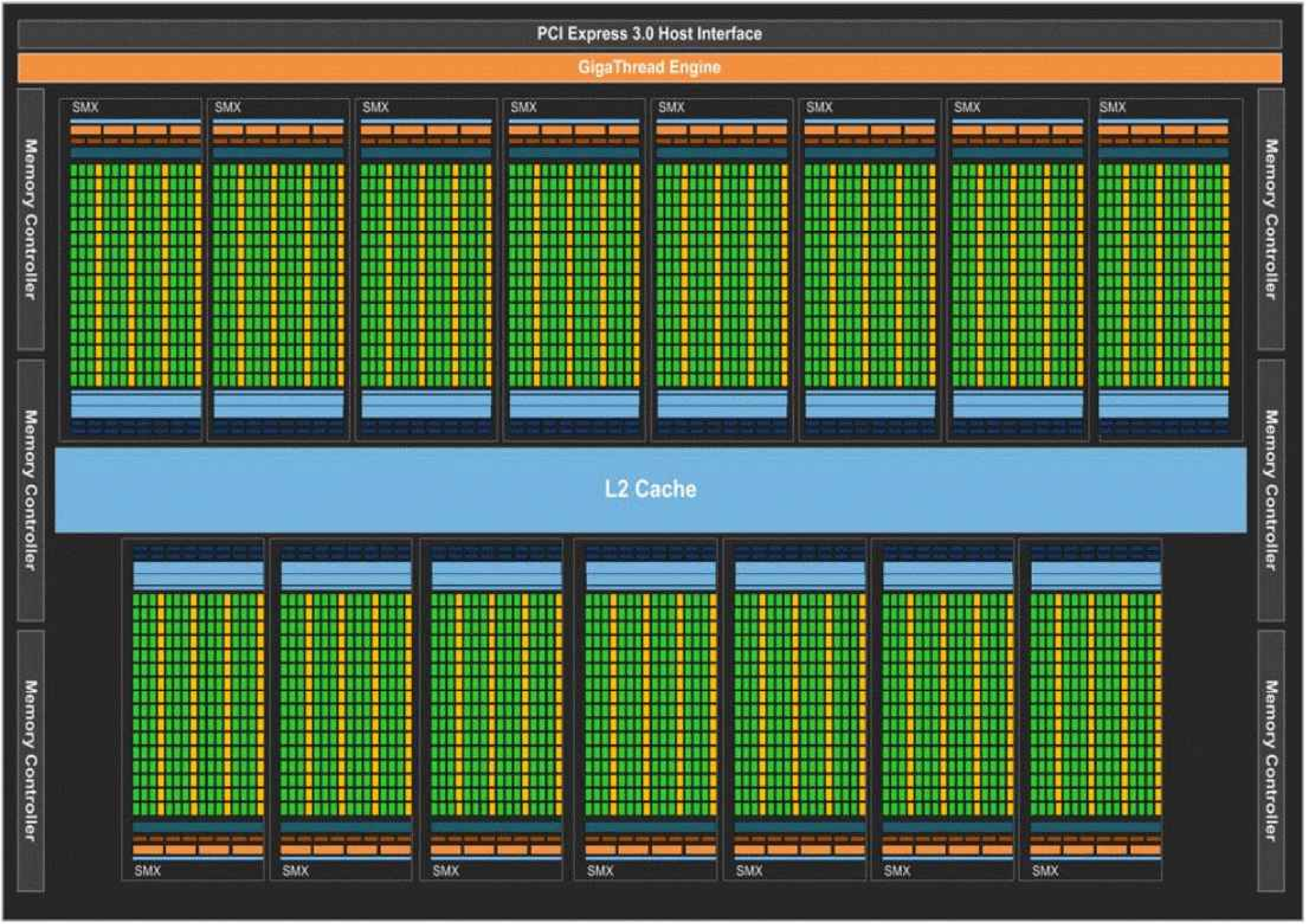


Performance:

Intel E5-2697 (Q3'13): SP → 0.518 Tflops, DP→ 0.259 Tflops
Nvidia K40 (Q4'13): SP→ 4.29 Tflops, DP→ 1.43 Tflops

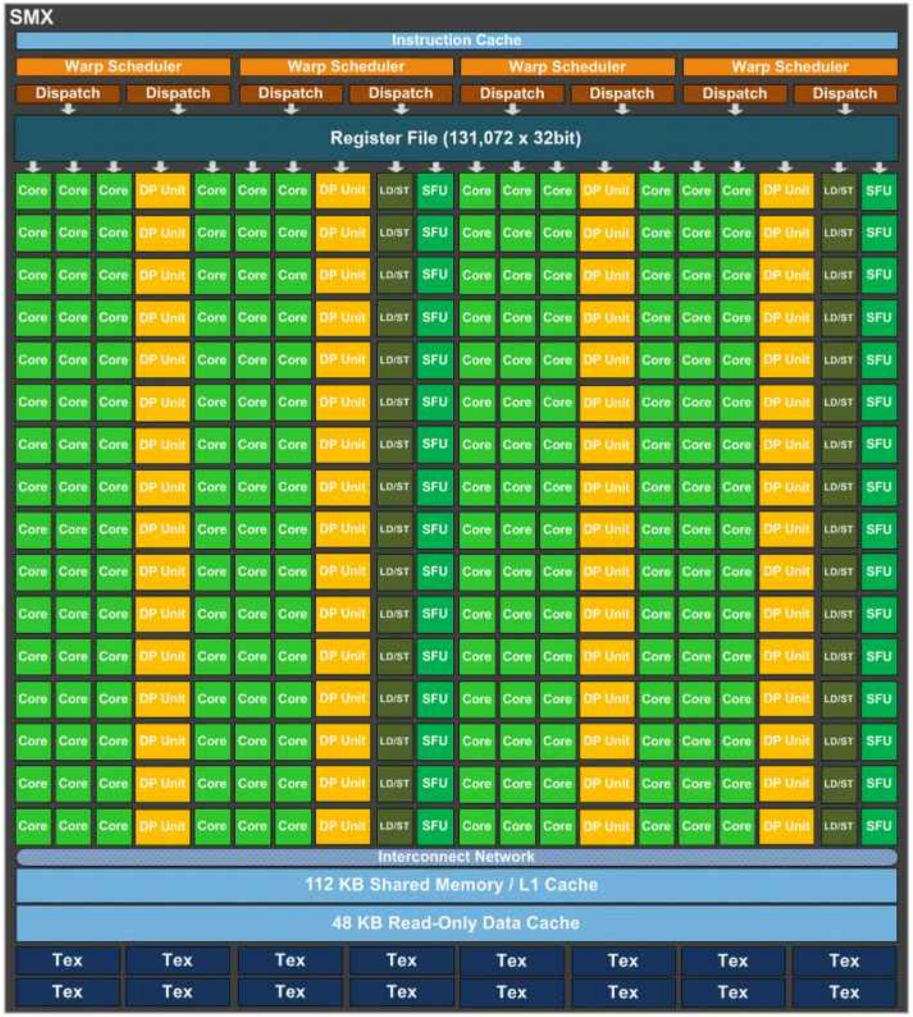
GPU Achitecture

- Streaming Multiprocessor
  - Perform the actual computations
  - Each SM has its own: control units, registers, execution pipelines, caches
- Global Memory
  - Similar to standard DRAM
  - Accessible from both CPU and GPU


PCI Express 3.0 Host Interface
GigaThread Engine
Memory Controller
Memory Controller
Memory Controller
Memory Controller
Memory Controller
Memory Controller
SMX
SMX
SMX
SMX
SMX
SMX
SMX
SMX
L2 Cache
SMX
SMX
SMX
SMX
SMX
SMX
SMX

GPU Architecture

- **Core/DP Units**: SP and DP arithmetic logic units.
- **Load/Store Units**: calculate source and destination addresses for 16 threads per clock. Load and store the data from/to cache or DRAM.
- **Special Functions Units (SFUs)**: Execute transcendental instructions such as *sin, cosine, reciprocal, and square root*. Each SFU executes one instruction per thread, per clock.


SMX
Instruction Cache
Warp Scheduler
Warp Scheduler
Warp Scheduler
Warp Scheduler
Dispatch
Dispatch
Dispatch
Dispatch
Dispatch
Dispatch
Dispatch
Dispatch
Register File (131,072 x 32bit)
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Interconnect Network
112 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex Tex Tex Tex Tex Tex Tex Tex
Tex Tex Tex Tex Tex Tex Tex Tex

Fonte: NVIDIA Whitepaper: NVIDIA Next Generation CUDA Compute Architecture: Kepler GK110/210 V1.0. 2014.

Architecture details

Here’s your new toy!

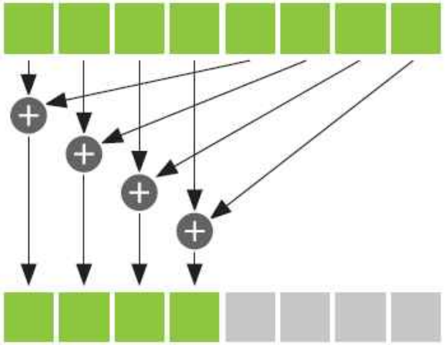
Two GPUs (GK210) per device
12GB RAM per GPU
480GB/s memory bandwidth

15 **Multiprocessors** (MP),
192 **CUDA Cores**/MP = 2880 CUDA Cores

~ 500-800 Mhz Clocks
~ 250 W

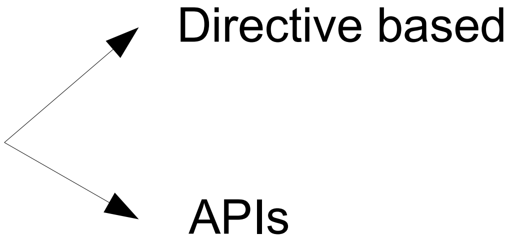
When/How to use them?

- Separate device memory
- Many-cores
- Multi-threading
- Vectors
- Memory strides matter
- Smaller caches



How to use them?

- Today
  - Libraries
  - Programming languages



Directive based

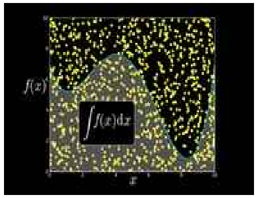
APIs

- Tomorrow
  - **Standard and HW agnostic** compiler directives
  - Unified architecture, standardized programming language
  - PGAS?

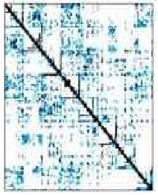
How to deal with it?

First rule: do not reinvent the wheel!

NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

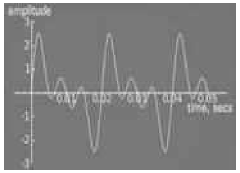
Vector Signal Image Processing

CULA tools

GPU Accelerated Linear Algebra

MAGMA

Matrix Algebra on GPU and Multicore



NVIDIA cuFFT

ROGUE WAVE SOFTWARE

IMSL Library

ArrayFire Matrix Computations

CUSP

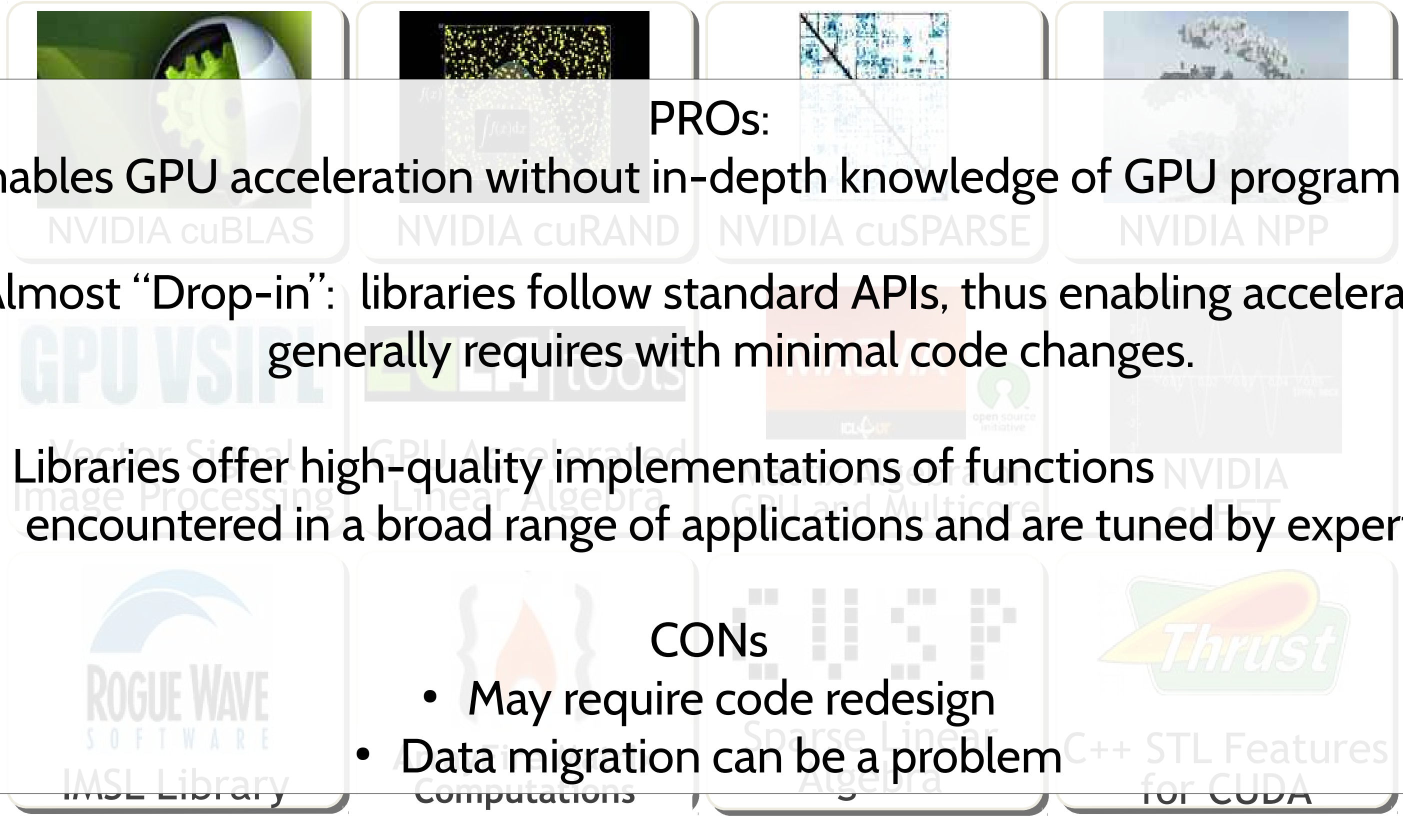
Sparse Linear Algebra

Thrust

C++ STL Features for CUDA

How to deal with it?

First rule: do not reinvent the wheel!



PROs:

- enables GPU acceleration without in-depth knowledge of GPU programming
- Almost "Drop-in": libraries follow standard APIs, thus enabling acceleration generally requires with minimal code changes.
- Libraries offer high-quality implementations of functions encountered in a broad range of applications and are tuned by experts

CONs

- May require code redesign
- Data migration can be a problem

GPU Programming

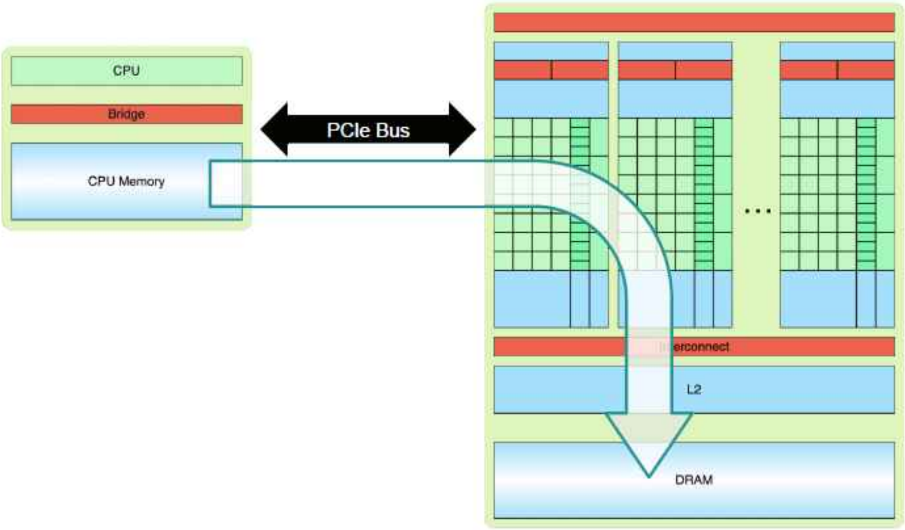
C	OpenACC, CUDA C, OpenCL, ...
C++	Thrust, CUDA C++, OpenCL, ...
Fortran	OpenACC, CUDA Fortran
Python	PyCUDA, Copperhead
Numerical analytics	MATLAB, Mathematica, LabVIEW

CUDA

- Only Nvidia GPUs (*)
- Freely available and well documented
- A lot of stuff on the web
- Mature project, but still evolving
- Set of **extensions** to **C/C++** to define the kernels and to configure the kernel execution
- Proprietary software

Heterogeneous programming

- CPU and GPU are **separate devices** with **separate memory** spaces
- Different portions of the code runs **on the CPU or on the GPU**.



Heterogeneous programming

“**host**”: the CPU/DRAM

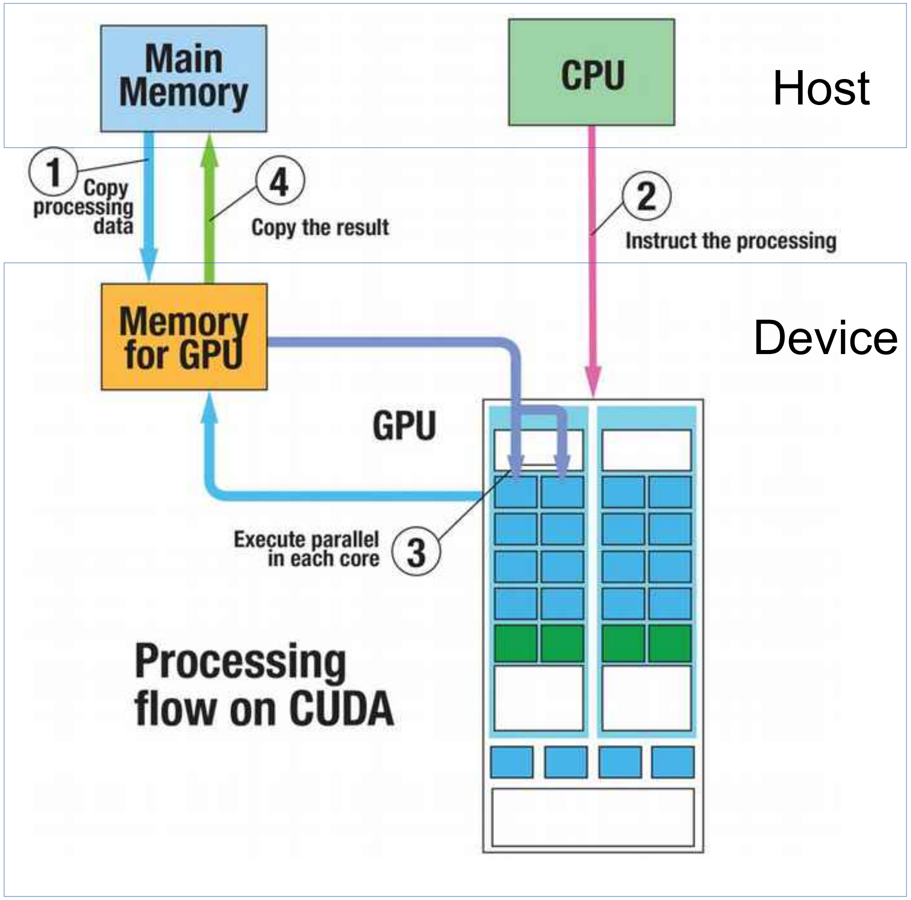
“**device**”: the GPU/and its memory

“**kernel**”: the chunk of your code that you invoke many times: once per input element.

Heterogeneous programming

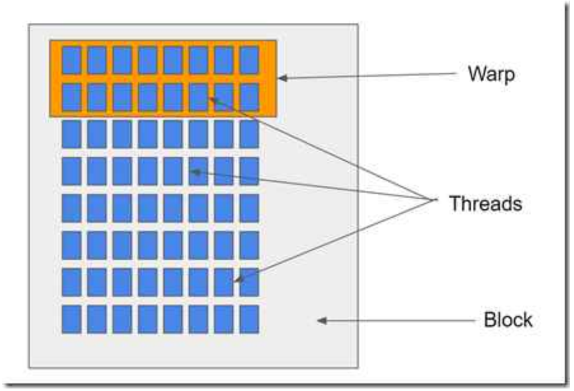
- Typical code progression
  - Memory allocated on host and device
  - Data is transferred **from** the *Host* **to** the *Device*
  - Kernel is lunched by the Host on the Device
  - Data is transferred **from** the *Device* **to** the *Host*.
  - Memory is deallocated.

Heterogeneous programming



http://rtcmagazine.com/articles/view/103718

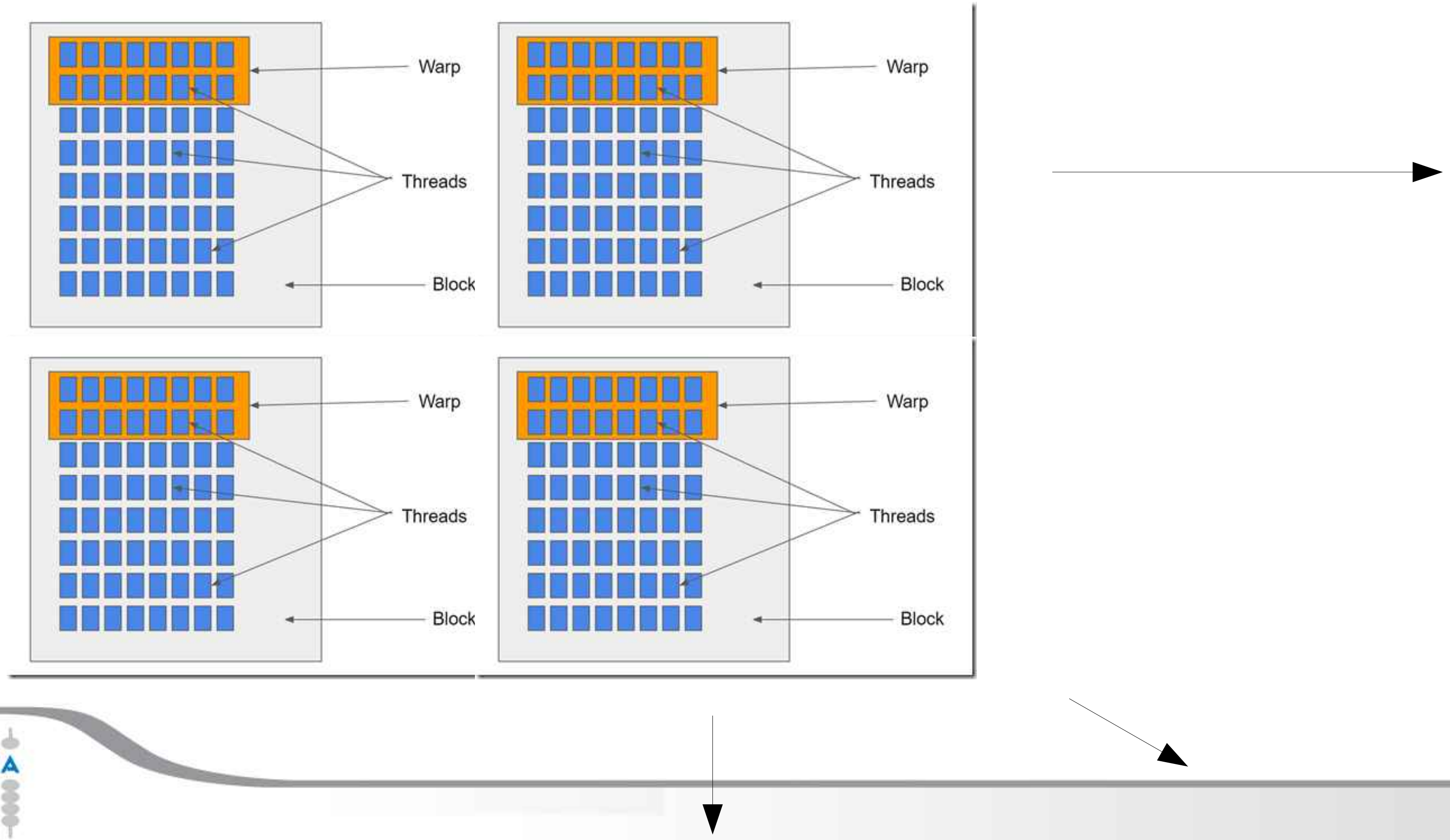
CUDA Execution Model



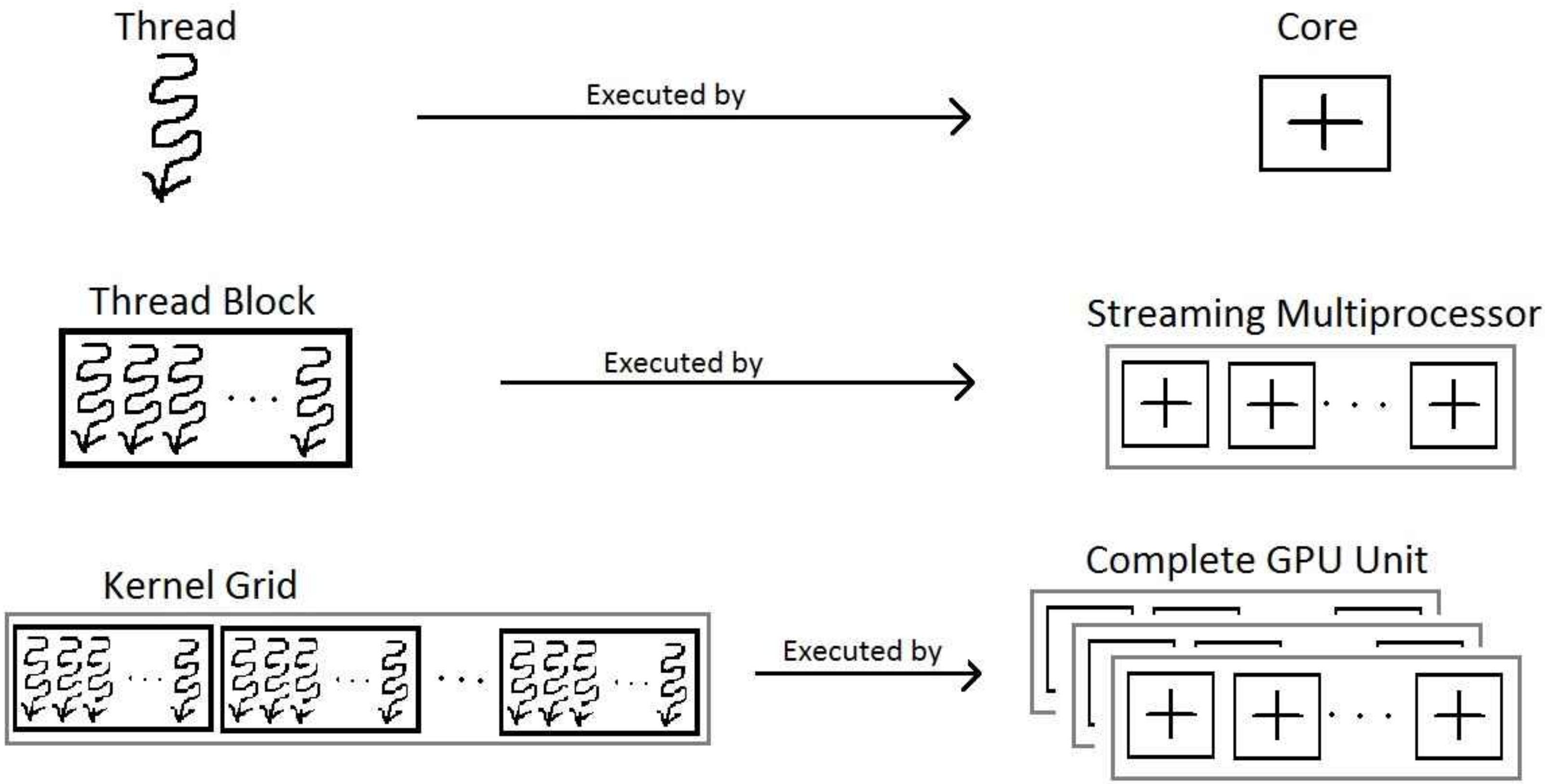
- **Warp**: group of 32 threads handled by the scheduler. Always use 32*n!
- **Thread**: each execute the kernel. Can be synchronized. Can be arranged in 3D: x,y,z. Useful for programming and memory access.
- **Block**: Group of threads.
- **Grids**: Group of Blocks

CUDA Execution Model

- maximum number of threads per dimension in a block is 1024(*)!



CUDA Execution Model

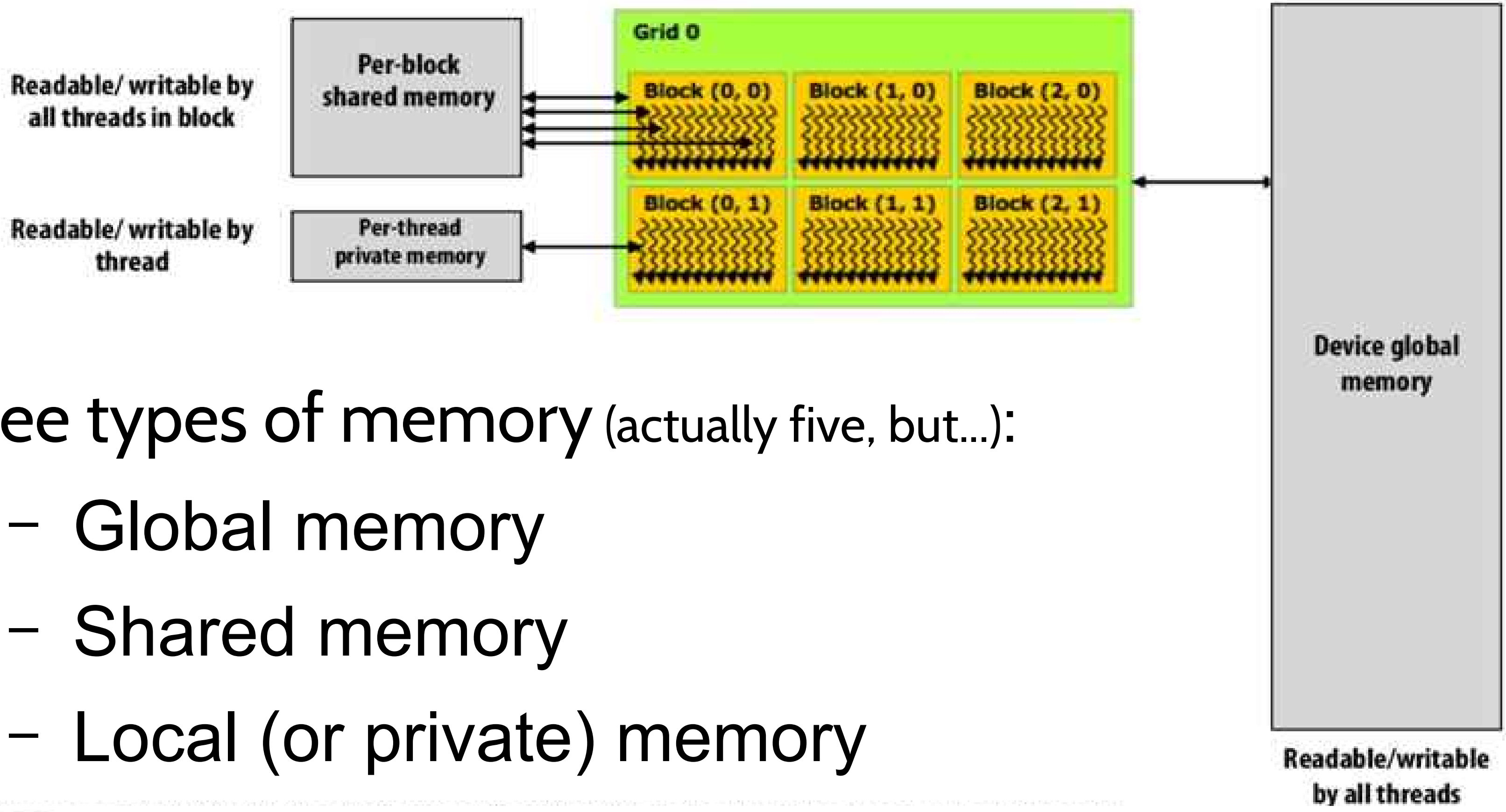


CUDA Execution Model

- Where do I find all these info?!

```
$CUDA_HOME/samples/bin/x86_64/
linux/release/deviceQuery
```

CUDA Memory


Readable/ writable by all threads in block
Per-block shared memory
Readable/ writable by thread
Per-thread private memory
Grid 0
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Device global memory
Readable/writable by all threads

Three types of memory (actually five, but...):

- Global memory
- Shared memory
- Local (or private) memory

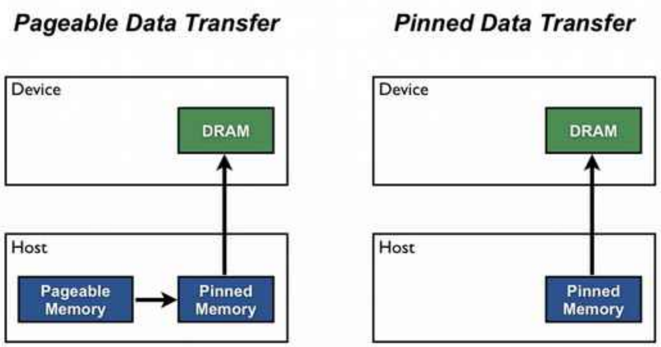
CUDA Memory

On the GPU:

- Memory optimization is vital on GPU.
- Different memory types have different latency.
- Coalescent access is mandatory.

On the DRAM:

- Pinned memory
- Pointers with restricted



If the memory addressed by the restrict-qualified pointer is modified, no other pointer will access that same memory.

CUDA Memory

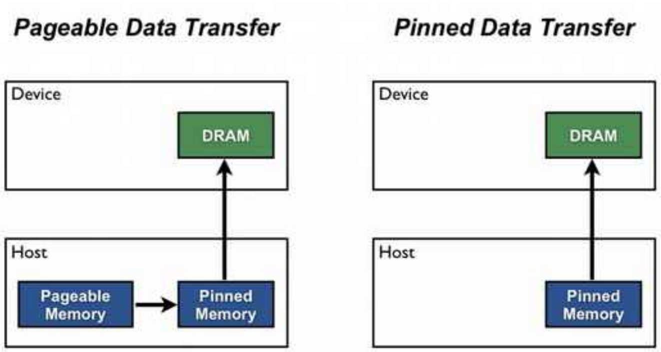
On the GPU:

- Memory optimization is vital on GPU.
- Different memory types have different latency.
- Coalescent access is mandatory.

ADVANCED!!

On the DRAM:

- Pinned memory
- Pointers with restricted



If the memory addressed by the restrict-qualified pointer is modified, no other pointer will access that same memory.

CUDA syntax

- A CUDA kernel function is defined using the

```
__global__
```

- A CUDA kernel always returns **void!**
- when a CUDA kernel is called, it is executed **N times** in parallel by N different CUDA threads on **one** device.
- CUDA threads that execute that kernel are specified using the **kernel execution** configuration **syntax**:

```
CudaKernelFunction <<<…,…>>> (arg_1, arg_2,…, arg_n)
```

CUDA syntax

- each thread has a unique thread ID, threads within a block can be synchronized
- the thread ID is accessible through the built-in variable

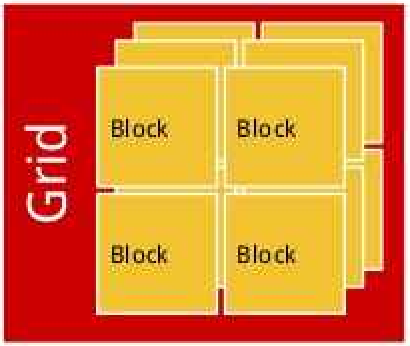
`threadIdx`

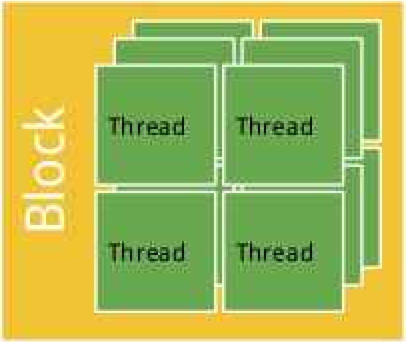
- `threadIdx` are a 3-component vector
  use `.x, .y, .z` to access its components

https://www.slideshare.net/pipatmet/hpp-week-1-summary

CUDA syntax

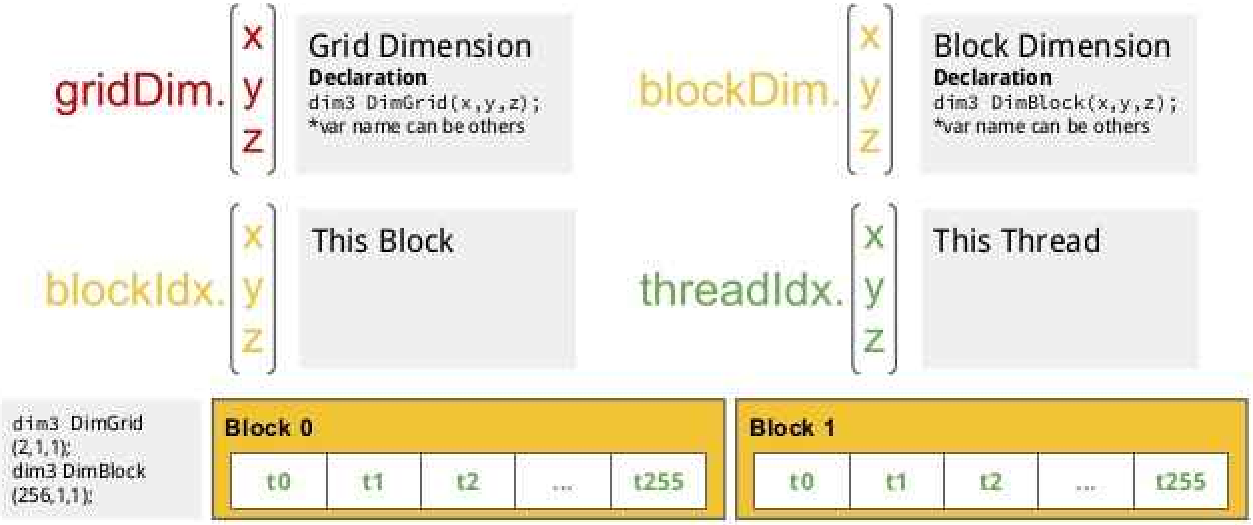
- **Grid** = [Vector~3D Matrix] of Blocks
  - **Block** = [Vector~3D Matrix] of Threads
    - **Thread** = One that computes





https://www.slideshare.net/pipatmet/hpp-week-1-summary

CUDA syntax



```
dim3 DimGrid(x,y,z);
```

*var name can be others

```
dim3 DimBlock(x,y,z);
```

*var name can be others

```
dim3 DimGrid
(2,1,1);
dim3 DimBlock
(256,1,1);
```

```
kernel_name<<<int,int>>> (args);
kernel_name<<<dim3,dim3>>> (args);

<<<gridDim, blockDim>>>
```

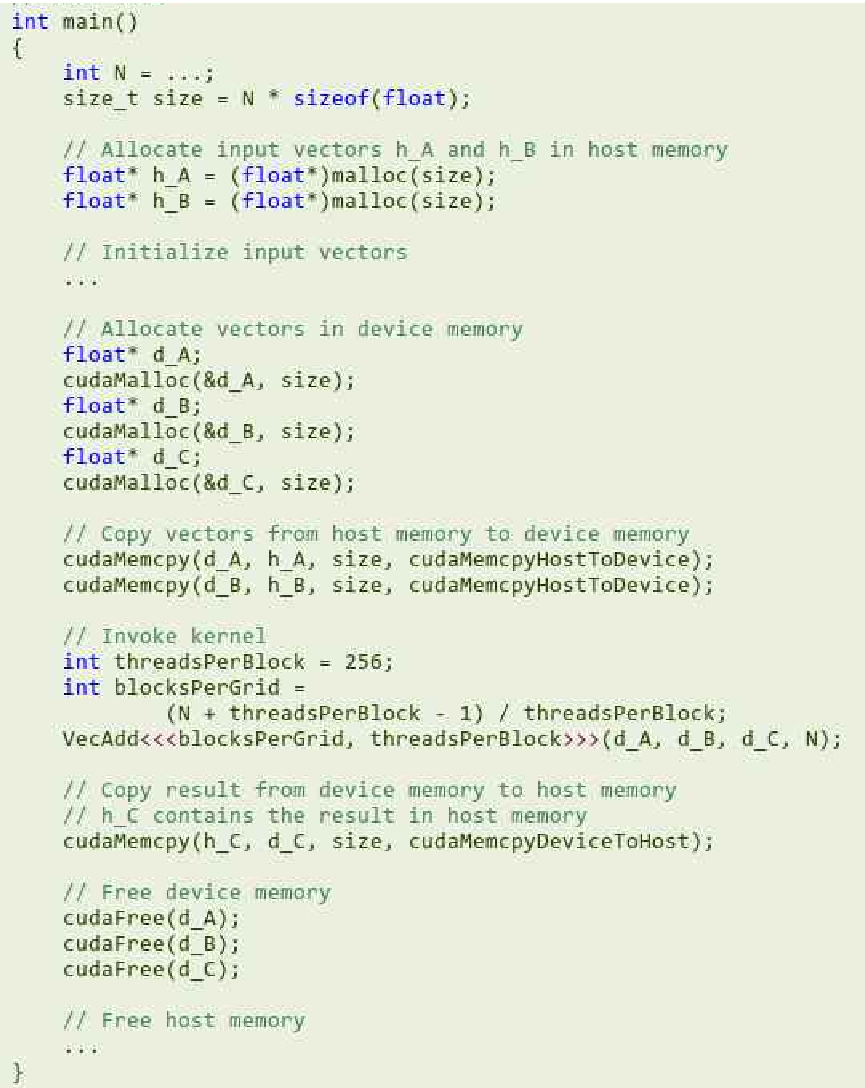
Imgs from: https://www.slideshare.net/pipatmet/hpp-week-1-summary

CUDA syntax

- “Standard” memory access
  - cudaMalloc
  - cudaFree
  - cudaMemcpy
- Unified memory access
  - cudaMallocManaged
  - cudaFree

CUDA syntax

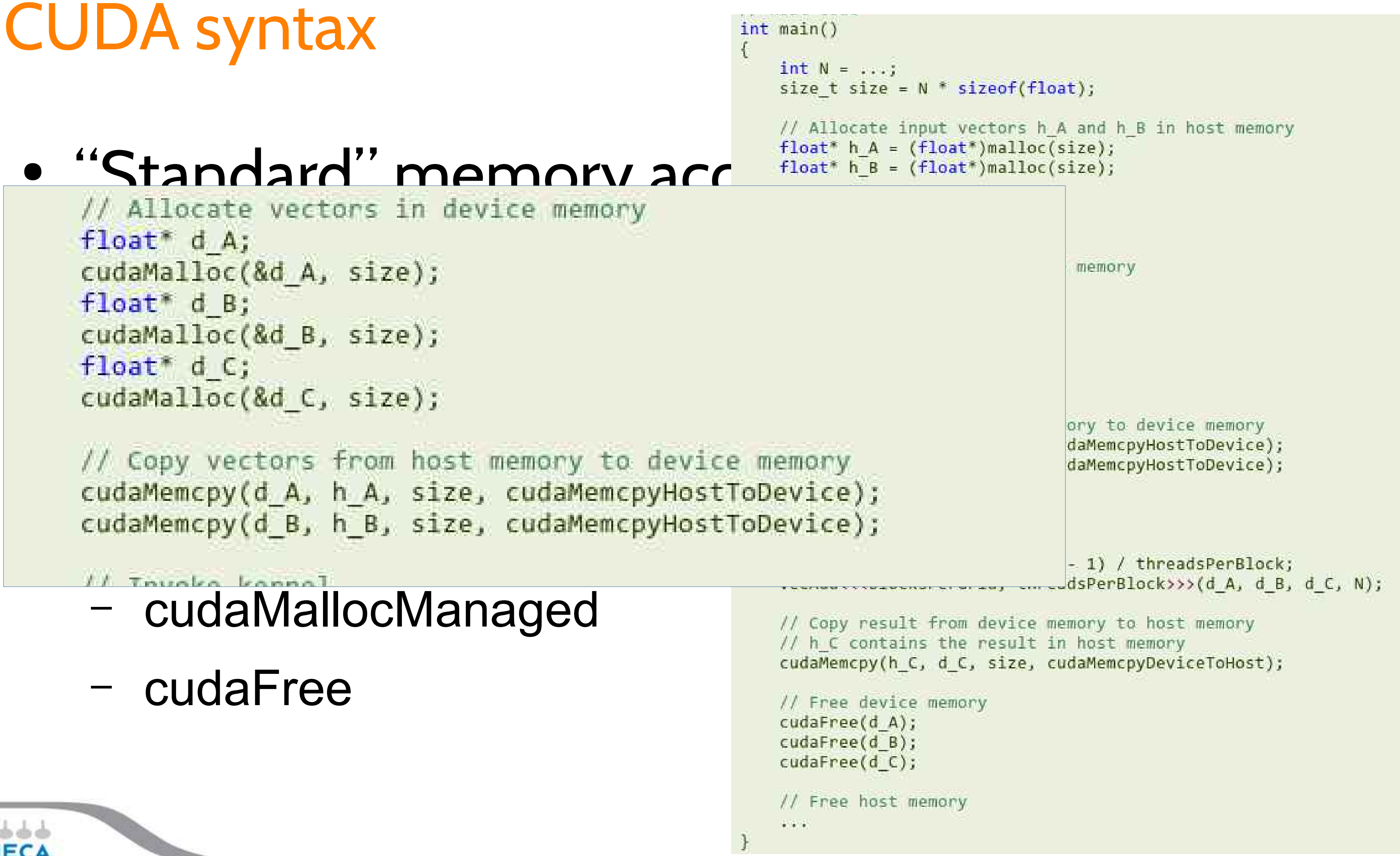
- “Standard” memory access
  - cudaMalloc
  - cudaFree
  - cudaMemcpy
    - Sync! Wait until kernel is finished, no need for manual sync
- Unified memory access
  - CudaMallocManaged
    - Needs cudaDeviceSync
  - cudaFree



```
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

CUDA syntax

- “Standard” memory acc



```
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

 memory

ory to device memory
daMemcpyHostToDevice);
daMemcpyHostToDevice);

- 1) / threadsPerBlock;
dsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

```
    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
```

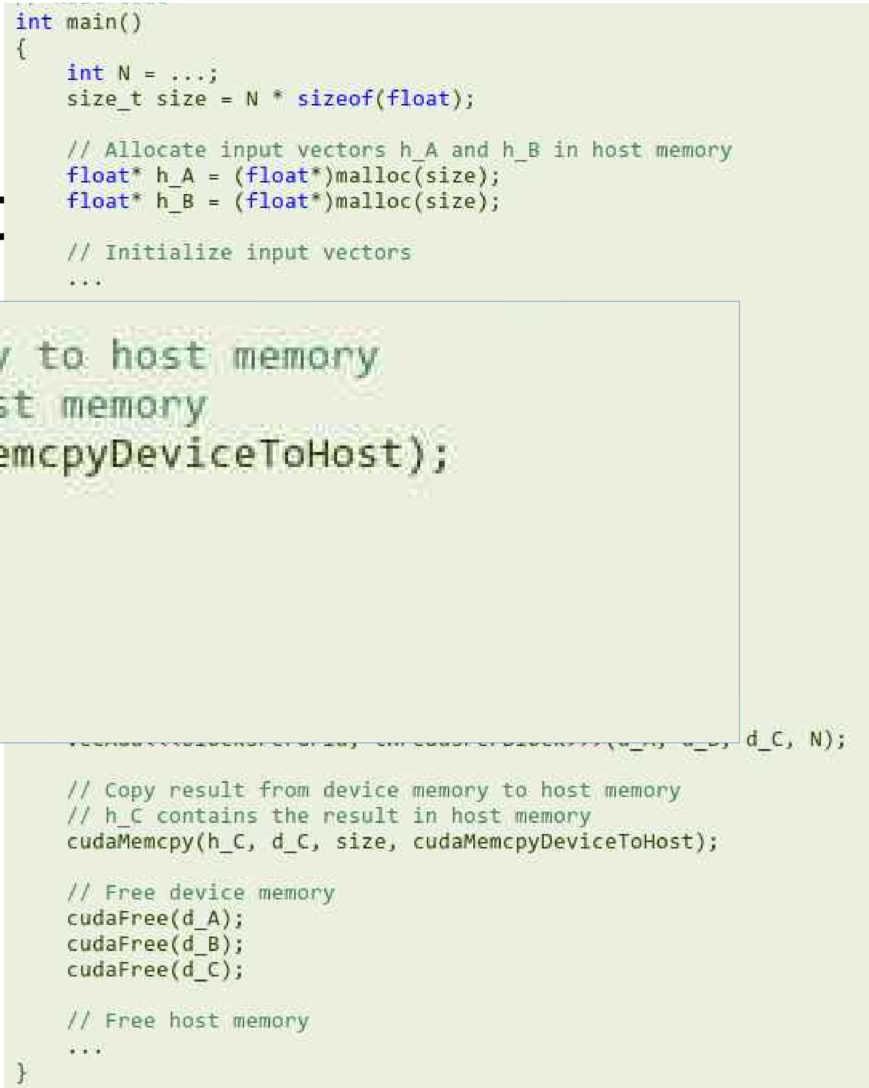
  - cudaMallocManaged
  - cudaFree

CUDA syntax

- “Standard” memory acc

```
// Copy result from device memory to host memory
// h_C contains the result in host memory
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

// Free device memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

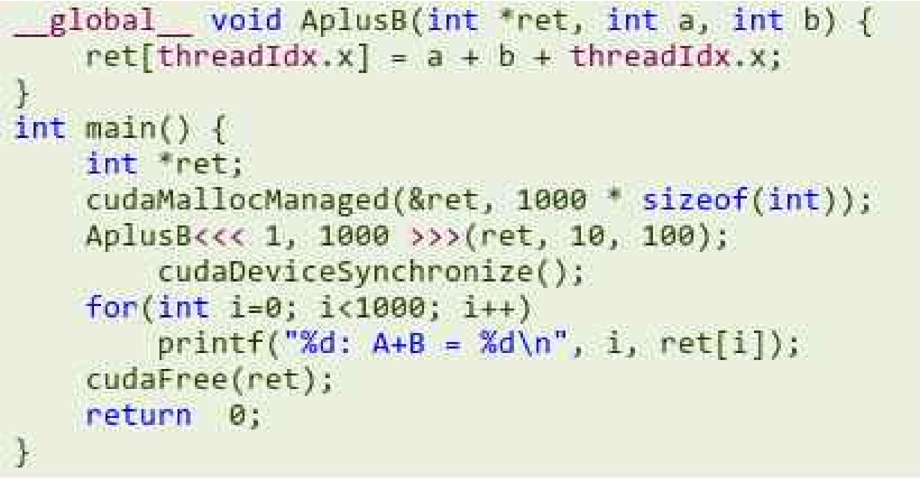
  - cudaMallocManaged
  - cudaFree



```
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

                                                      d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

CUDA syntax

- “Standard” memory access
  - cudaMalloc
  - cudaFree
  - cudaMemcpy
- Unified memory acces
  - cudaMallocManaged
  - cudaFree



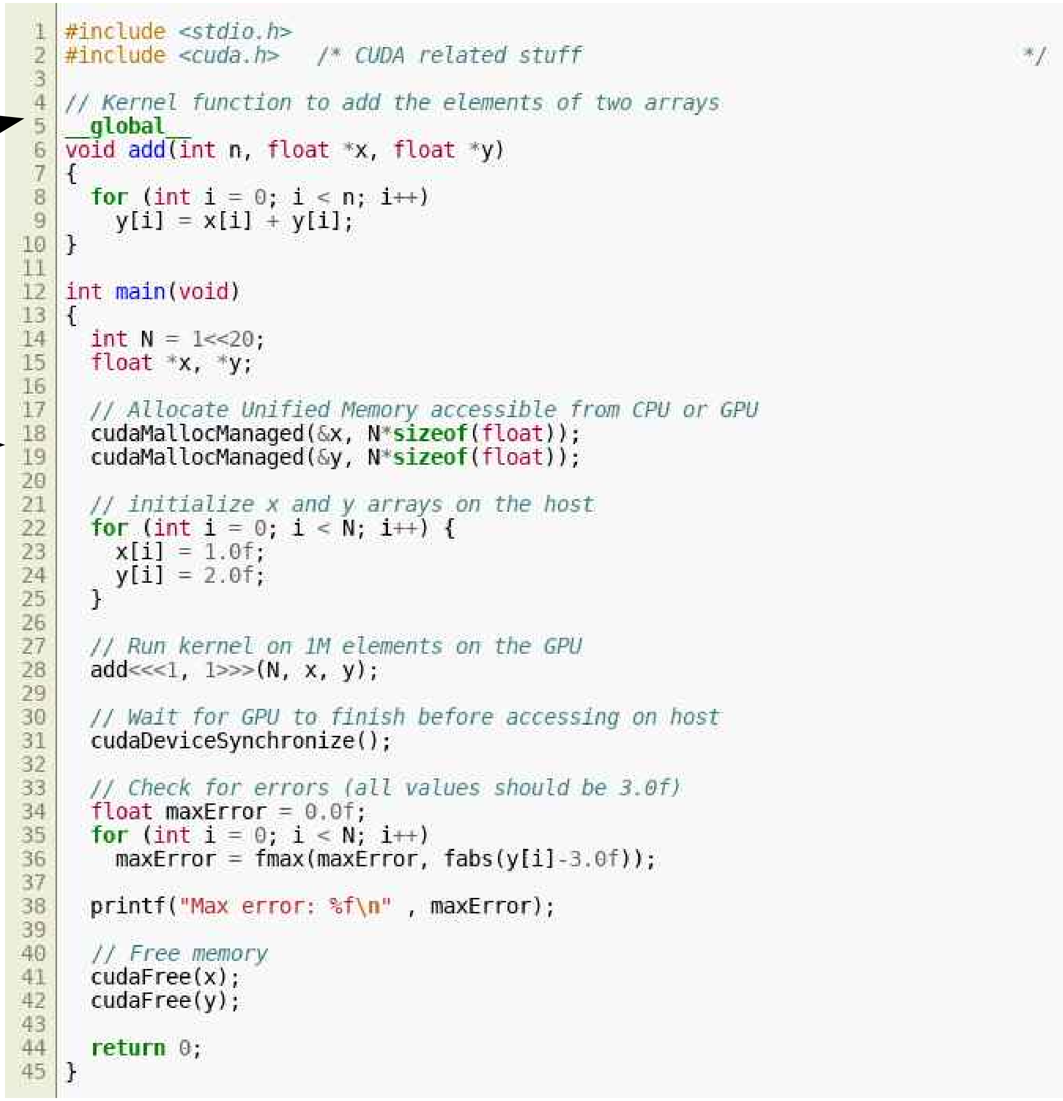
```
__global__ void AplusB(int *ret, int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
    int *ret;
    cudaMallocManaged(&ret, 1000 * sizeof(int));
    AplusB<<< 1, 1000 >>>(ret, 10, 100);
        cudaDeviceSynchronize();
    for(int i=0; i<1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    cudaFree(ret);
    return  0;
}
```

Your first (?) kernel

Kernel that runs on the GPU must return **void!**

Memory allocation and movements handled by CUDA API

https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/



```
#include <stdio.h>
#include <cuda.h>    /* CUDA related stuff                                                  */

// Kernel function to add the elements of two arrays
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
    y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20;
  float *x, *y;

  // Allocate Unified Memory accessible from CPU or GPU
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the GPU
  add<<<1, 1>>>(N, x, y);

  // Wait for GPU to finish before accessing on host
  cudaDeviceSynchronize();

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));

  printf("Max error: %f\n" , maxError);

  // Free memory
  cudaFree(x);
  cudaFree(y);

  return 0;
}
```

Example 1

- Adding matrices

$$\mathbf{A}+\mathbf{B}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \cdots & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{22}+b_{22} & \cdots & a_{2n}+b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}+b_{m1} & a_{m2}+b_{m2} & \cdots & a_{mn}+b_{mn} \end{bmatrix}$$

Example 2

- Calculating pi **(THE WRONG WAY)**

$$\begin{aligned}\frac{\pi}{4} &= \arctan(1)\\ &= \int_0^1 \frac{1}{1+x^2}\,dx\\ &= \int_0^1 \left(\sum_{k=0}^{n} (-1)^k x^{2k} + \frac{(-1)^{n+1}\,x^{2n+2}}{1+x^2}\right) dx\\ &= \left(\sum_{k=0}^{n} \frac{(-1)^k}{2k+1}\right) + (-1)^{n+1}\left(\int_0^1 \frac{x^{2n+2}}{1+x^2}\,dx\right).\end{aligned}$$

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

Example 3

- Calculating pi **(THE RIGHT WAY)**

$$\begin{aligned}\frac{\pi}{4} &= \arctan(1)\\ &= \int_0^1 \frac{1}{1+x^2}\,dx\end{aligned}$$

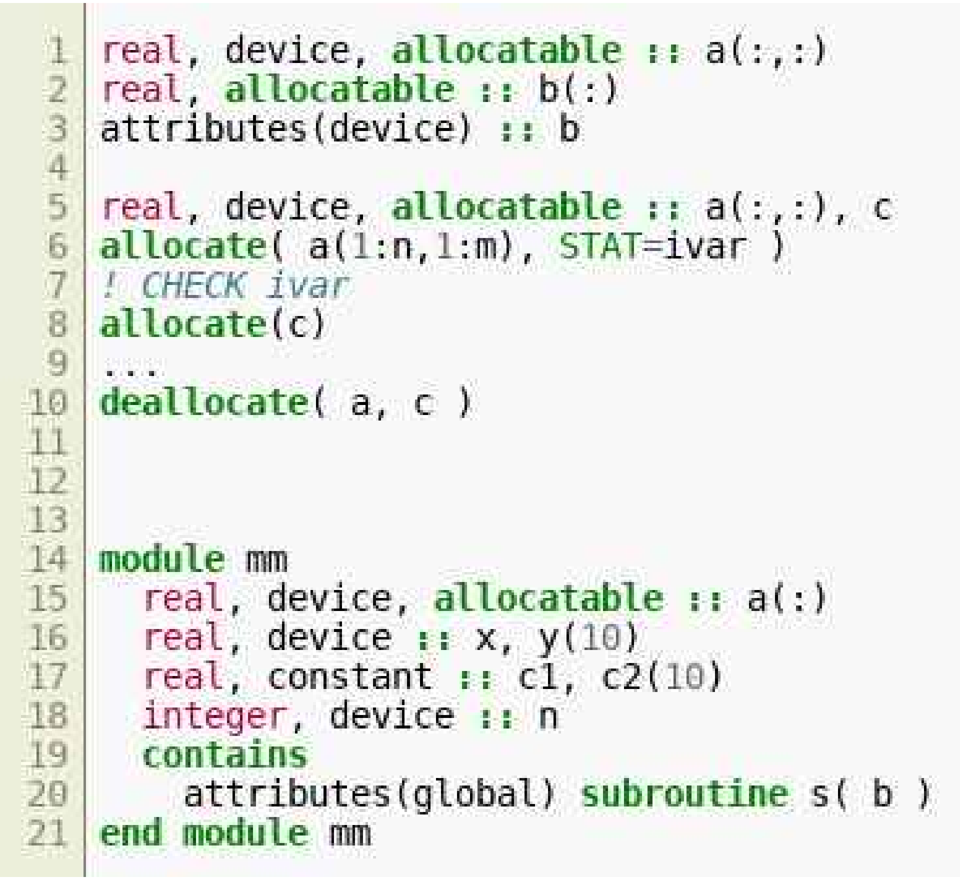
$$\int_a^b f(x)\,dx \approx h\sum_{n=0}^{N-1} f(x_n) \qquad x_n = a + nh \qquad h = (b-a)/N$$

CUDA Fortran

- Fortran analog to CUDA C
- Syntax is similar to CUDA, but more concise
- Complete syntax only on PGI compilers (16.10 community edition freely available)
- Partial implementation on IBM compilers

CUDA Fortran Syntax

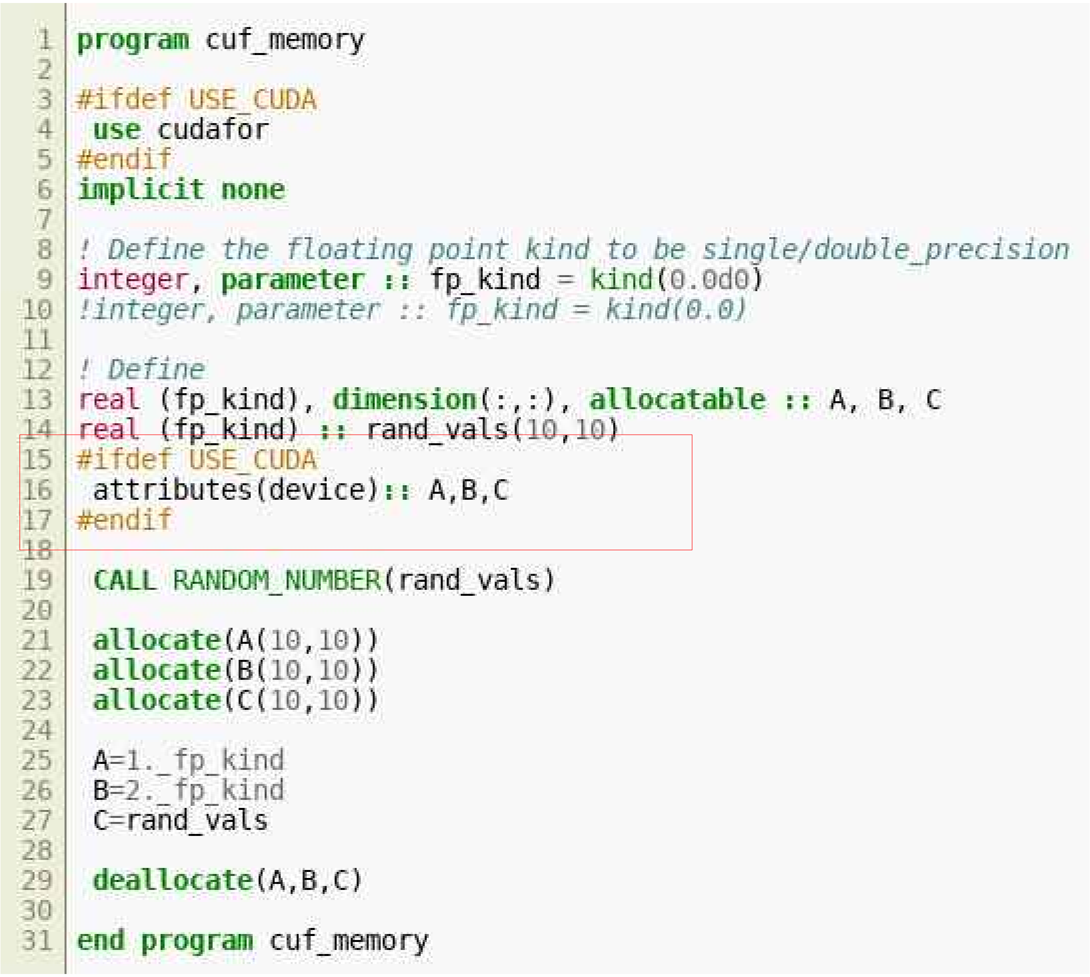
- Allocate done by host, according to *“device”* attribute
- Memory is not virtual! You may run out, check!
- Just copy (no need for cuda sync).



```
real, device, allocatable :: a(:,:)
real, allocatable :: b(:)
attributes(device) :: b

real, device, allocatable :: a(:,:), c
allocate( a(1:n,1:m), STAT=ivar )
! CHECK ivar
allocate(c)
...
deallocate( a, c )



module mm
  real, device, allocatable :: a(:)
  real, device :: x, y(10)
  real, constant :: c1, c2(10)
  integer, device :: n
  contains
    attributes(global) subroutine s( b )
end module mm
```

CUDA Fortran Syntax

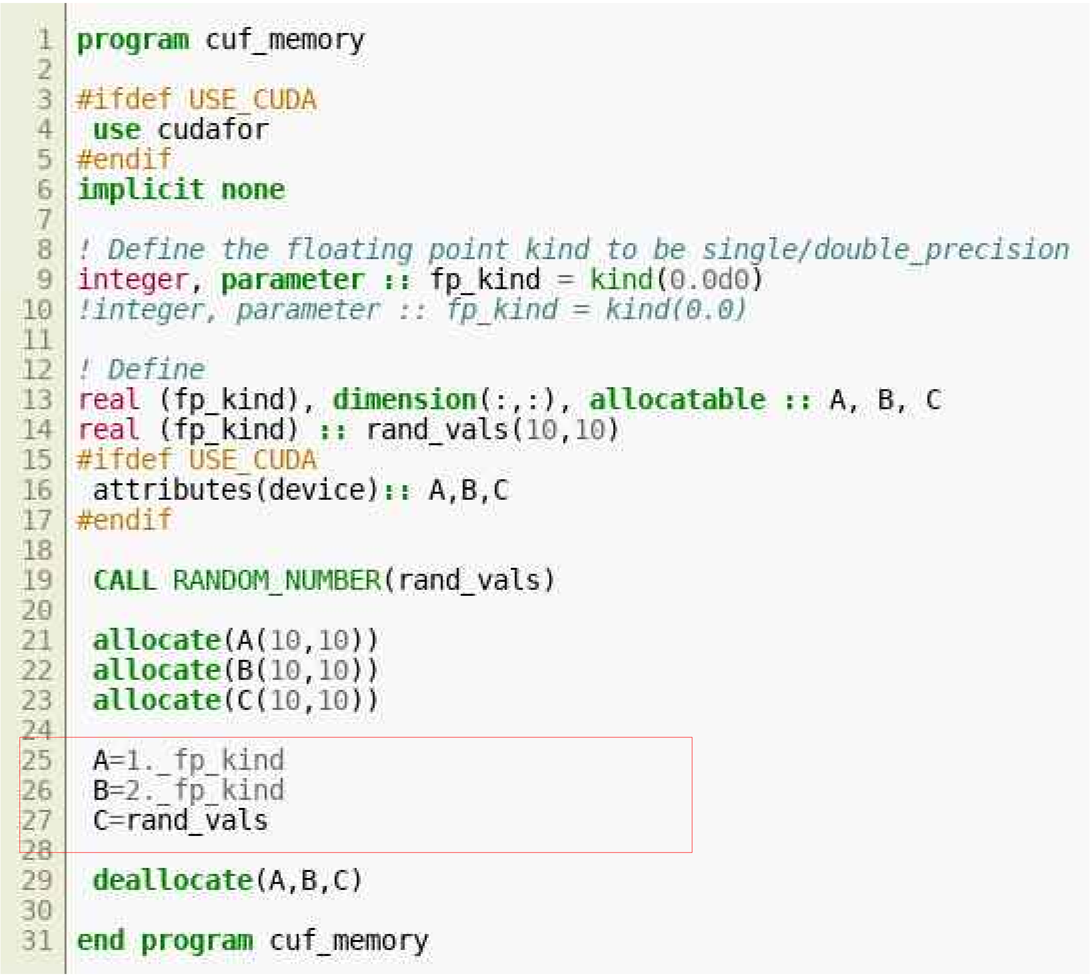
- Allocate done by host, according to *"device"* attribute
- Memory is not virtual! You may run out, check!
- Just copy (no need for cuda sync)



```
program cuf_memory

#ifdef USE_CUDA
 use cudafor
#endif
implicit none

! Define the floating point kind to be single/double_precision
integer, parameter :: fp_kind = kind(0.0d0)
!integer, parameter :: fp_kind = kind(0.0)

! Define
real (fp_kind), dimension(:,:), allocatable :: A, B, C
real (fp_kind) :: rand_vals(10,10)
#ifdef USE_CUDA
 attributes(device):: A,B,C
#endif

 CALL RANDOM_NUMBER(rand_vals)

 allocate(A(10,10))
 allocate(B(10,10))
 allocate(C(10,10))

 A=1._fp_kind
 B=2._fp_kind
 C=rand_vals

 deallocate(A,B,C)

end program cuf_memory
```

CUDA Fortran Syntax

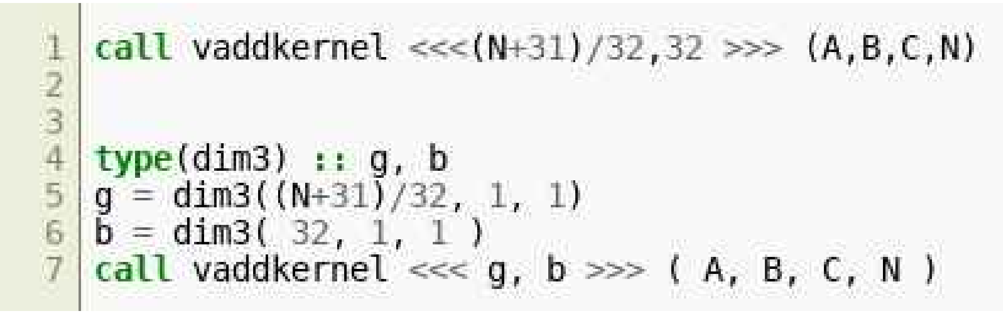
- Every copy statement is blocking
- Copy will wait until kernel has finished
- Scalars can be passed by value to kernels



```fortran
program cuf_memory

#ifdef USE_CUDA
 use cudafor
#endif
implicit none

! Define the floating point kind to be single/double_precision
integer, parameter :: fp_kind = kind(0.0d0)
!integer, parameter :: fp_kind = kind(0.0)

! Define
real (fp_kind), dimension(:,:), allocatable :: A, B, C
real (fp_kind) :: rand_vals(10,10)
#ifdef USE_CUDA
 attributes(device):: A,B,C
#endif

 CALL RANDOM_NUMBER(rand_vals)

 allocate(A(10,10))
 allocate(B(10,10))
 allocate(C(10,10))

 A=1._fp_kind
 B=2._fp_kind
 C=rand_vals

 deallocate(A,B,C)

end program cuf_memory
```

CUDA Fortran Syntax

- Running kernels

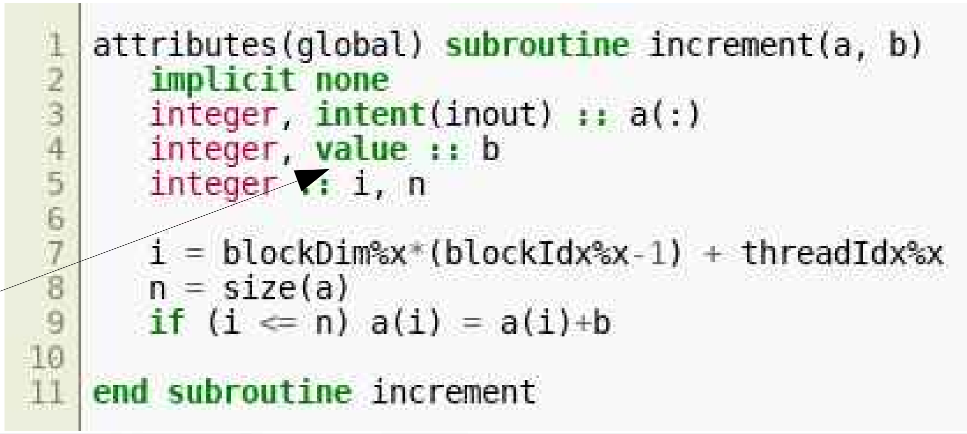


```
call vaddkernel <<<(N+31)/32,32 >>> (A,B,C,N)


type(dim3) :: g, b
g = dim3((N+31)/32, 1, 1)
b = dim3( 32, 1, 1 )
call vaddkernel <<< g, b >>> ( A, B, C, N )
```

You can create interfaces
Launch is **asynchronous!**
It will return immediately so be careful with timing.

CUDA Fortran Syntax

- Writing kernels:
  - ”global” attribute defines kernels
  - Scalars and fixed size arrays are in local memory
  - Allowed data types:
    - Integer(1..8), logical, real(4,8), **complex(4,8)**, derivedtype
  - Parameters by ***value***

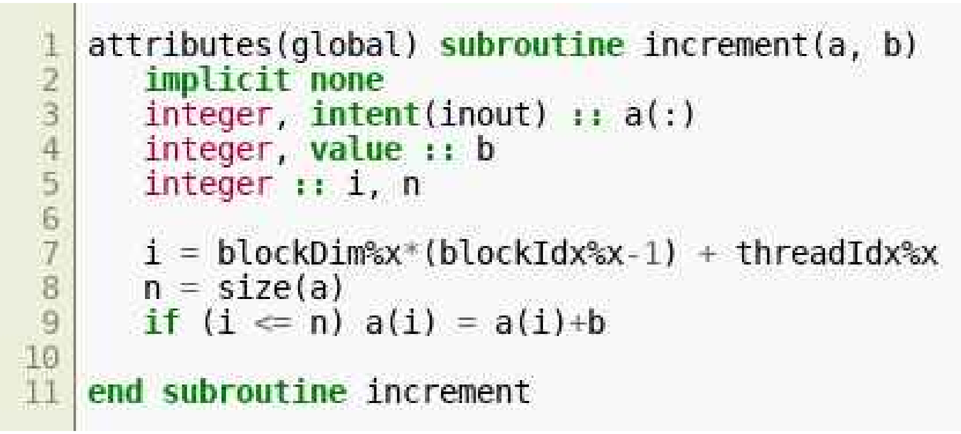


```
attributes(global) subroutine increment(a, b)
   implicit none
   integer, intent(inout) :: a(:)
   integer, value :: b
   integer :: i, n

   i = blockDim%x*(blockIdx%x-1) + threadIdx%x
   n = size(a)
   if (i <= n) a(i) = a(i)+b

end subroutine increment
```

!

CUDA Fortran Syntax

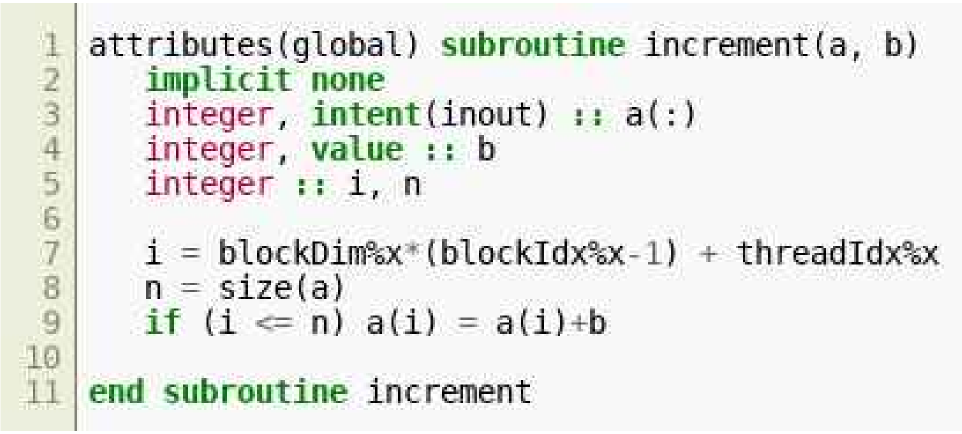
- Writing kernels:
  - Predefined variables:
    - blockIdx, threadIdx, gridDim, blockDim, warpSize
  - Valid statements
    - Assignment
    - For, do, while, if, goto, switch...
    - Call device function
    - Call intrinsic function



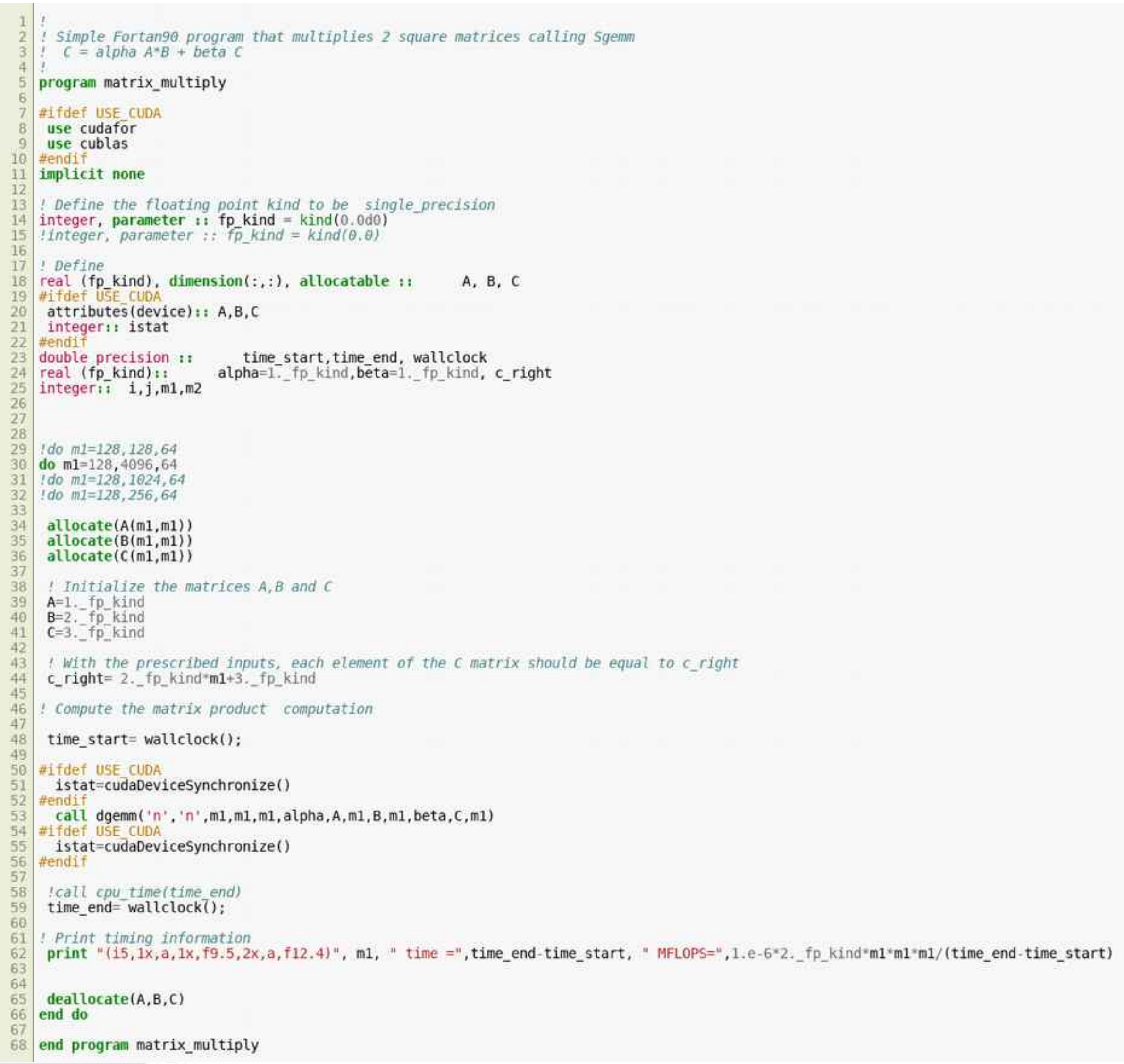
```fortran
attributes(global) subroutine increment(a, b)
   implicit none
   integer, intent(inout) :: a(:)
   integer, value :: b
   integer :: i, n

   i = blockDim%x*(blockIdx%x-1) + threadIdx%x
   n = size(a)
   if (i <= n) a(i) = a(i)+b

end subroutine increment
```

CUDA Fortran Syntax

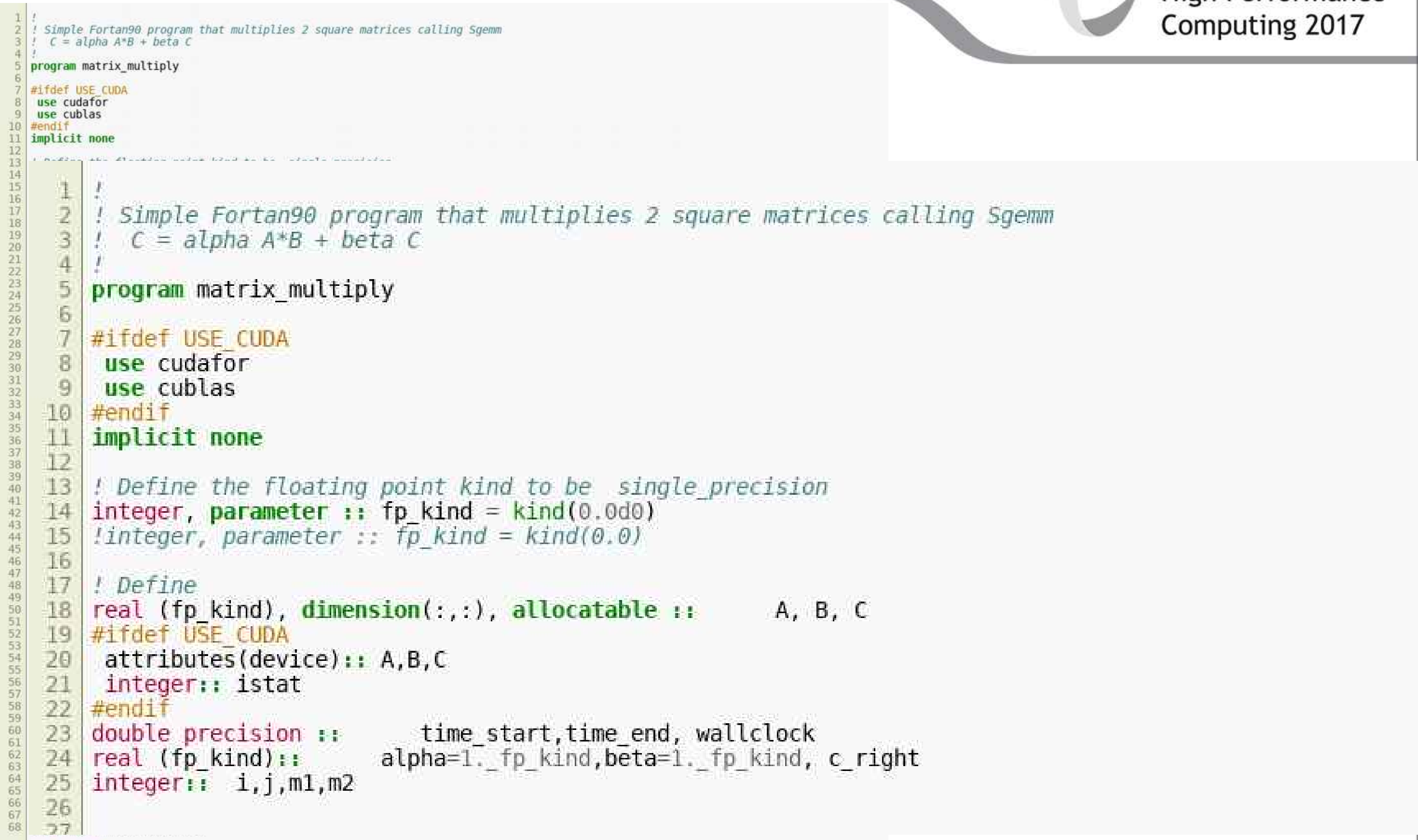
- Writing kernels:
  - INVALID statements
    - I/O (read, write, open…)
    - Pointer assignment
    - Recursive calls
    - ENTRY, ASSIGN statement
    - Stop, pause
    - (allocate/deallocate in PGI 13.0)



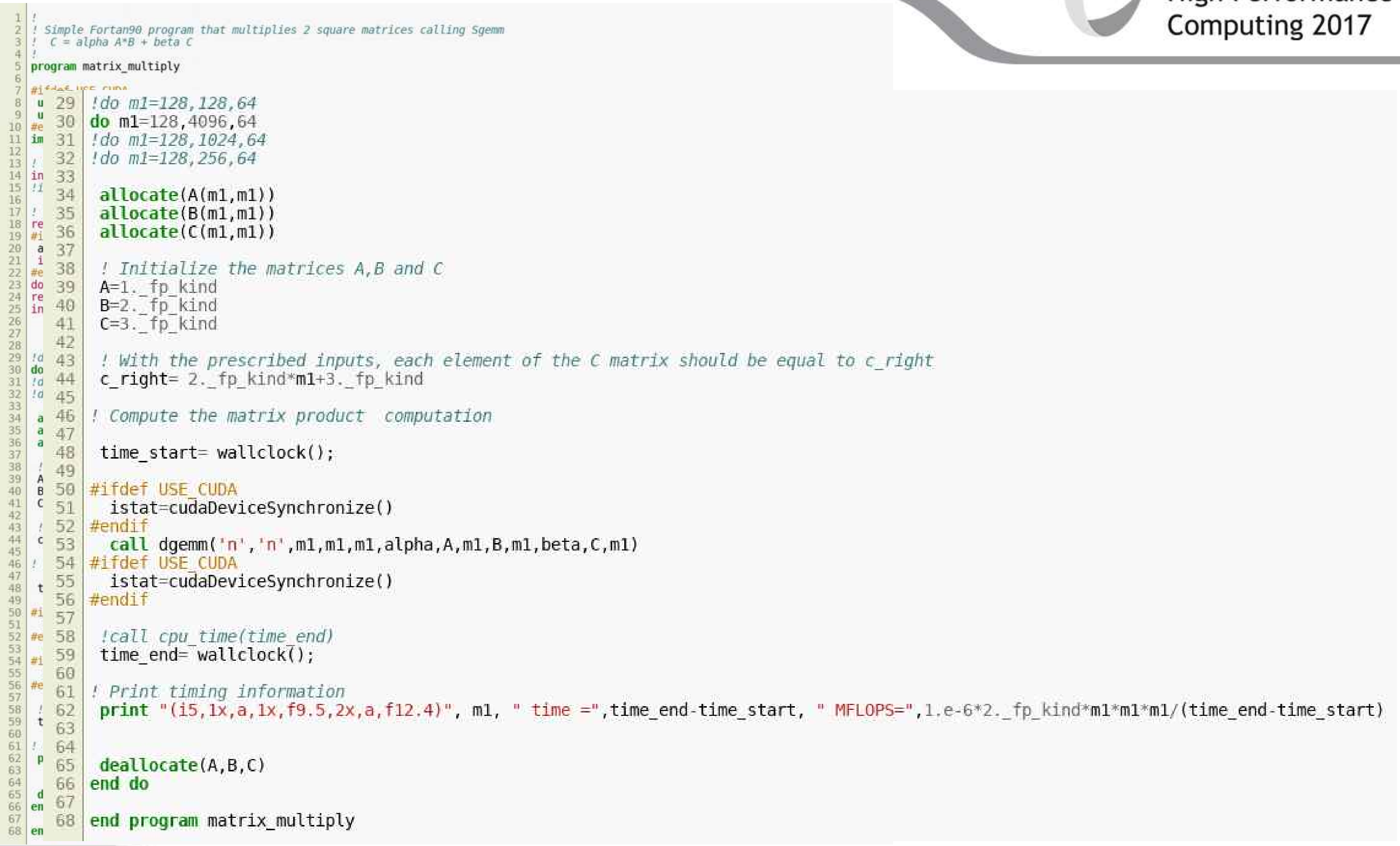
```fortran
attributes(global) subroutine increment(a, b)
  implicit none
  integer, intent(inout) :: a(:)
  integer, value :: b
  integer :: i, n

  i = blockDim%x*(blockIdx%x-1) + threadIdx%x
  n = size(a)
  if (i <= n) a(i) = a(i)+b

end subroutine increment
```



```fortran
!
! Simple Fortan90 program that multiplies 2 square matrices calling Sgemm
!  C = alpha A*B + beta C
!
program matrix_multiply

#ifdef USE_CUDA
 use cudafor
 use cublas
#endif
implicit none

! Define the floating point kind to be  single_precision
integer, parameter :: fp_kind = kind(0.0d0)
!integer, parameter :: fp_kind = kind(0.0)

! Define
real (fp_kind), dimension(:,:), allocatable ::      A, B, C
#ifdef USE_CUDA
 attributes(device):: A,B,C
 integer:: istat
#endif
double precision ::      time_start,time_end, wallclock
real (fp_kind)::      alpha=1._fp_kind,beta=1._fp_kind, c_right
integer::  i,j,m1,m2



!do m1=128,128,64
do m1=128,4096,64
!do m1=128,1024,64
!do m1=128,256,64

 allocate(A(m1,m1))
 allocate(B(m1,m1))
 allocate(C(m1,m1))

 ! Initialize the matrices A,B and C
 A=1._fp_kind
 B=2._fp_kind
 C=3._fp_kind

 ! With the prescribed inputs, each element of the C matrix should be equal to c_right
 c_right= 2._fp_kind*m1+3._fp_kind

! Compute the matrix product  computation

 time_start= wallclock();

#ifdef USE_CUDA
  istat=cudaDeviceSynchronize()
#endif
  call dgemm('n','n',m1,m1,m1,alpha,A,m1,B,m1,beta,C,m1)
#ifdef USE_CUDA
  istat=cudaDeviceSynchronize()
#endif

 !call cpu_time(time_end)
 time_end= wallclock();

! Print timing information
 print "(i5,1x,a,1x,f9.5,2x,a,f12.4)", m1, " time =",time_end-time_start, " MFLOPS=",1.e-6*2._fp_kind*m1*m1*m1/(time_end-time_start)


 deallocate(A,B,C)
end do

end program matrix_multiply
```



```fortran
!
! Simple Fortan90 program that multiplies 2 square matrices calling Sgemm
!   C = alpha A*B + beta C
!
program matrix_multiply

#ifdef USE_CUDA
 use cudafor
 use cublas
#endif
implicit none

! Define the floating point kind to be  single_precision
integer, parameter :: fp_kind = kind(0.0d0)
!integer, parameter :: fp_kind = kind(0.0)

! Define
real (fp_kind), dimension(:,:), allocatable ::      A, B, C
#ifdef USE_CUDA
 attributes(device):: A,B,C
 integer:: istat
#endif
double precision ::      time_start,time_end, wallclock
real (fp_kind)::      alpha=1._fp_kind,beta=1._fp_kind, c_right
integer::  i,j,m1,m2

```

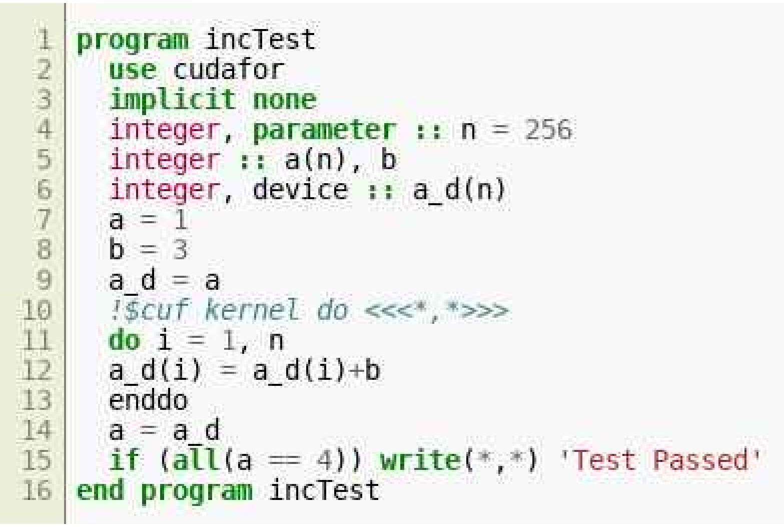


```
!
! Simple Fortan90 program that multiplies 2 square matrices calling Sgemm
!  C = alpha A*B + beta C
!
program matrix_multiply

```

```
!do m1=128,128,64
do m1=128,4096,64
!do m1=128,1024,64
!do m1=128,256,64

 allocate(A(m1,m1))
 allocate(B(m1,m1))
 allocate(C(m1,m1))

 ! Initialize the matrices A,B and C
 A=1._fp_kind
 B=2._fp_kind
 C=3._fp_kind

 ! With the prescribed inputs, each element of the C matrix should be equal to c_right
 c_right= 2._fp_kind*m1+3._fp_kind

! Compute the matrix product  computation

 time_start= wallclock();

#ifdef USE_CUDA
  istat=cudaDeviceSynchronize()
#endif
  call dgemm('n','n',m1,m1,m1,alpha,A,m1,B,m1,beta,C,m1)
#ifdef USE_CUDA
  istat=cudaDeviceSynchronize()
#endif

 !call cpu_time(time_end)
 time_end= wallclock();

! Print timing information
 print "(i5,1x,a,1x,f9.5,2x,a,f12.4)", m1, " time =",time_end-time_start, " MFLOPS=",1.e-6*2._fp_kind*m1*m1*m1/(time_end-time_start)


 deallocate(A,B,C)
end do

end program matrix_multiply
```

CUDA Fortran Syntax

- Cuf kernels, automatic kernel generation!



```
program incTest
  use cudafor
  implicit none
  integer, parameter :: n = 256
  integer :: a(n), b
  integer, device :: a_d(n)
  a = 1
  b = 3
  a_d = a
  !$cuf kernel do <<<*,*>>>
  do i = 1, n
  a_d(i) = a_d(i)+b
  enddo
  a = a_d
  if (all(a == 4)) write(*,*) 'Test Passed'
end program incTest
```

OpenACC

- Directive based
- Initiative to guide future OpenMP standardization
- Targets NVIDIA and AMD GPUs, Intel's Xeon Phi, FPGAs ...
- Works with C, C++ and Fortran
- Standard available at: www.openacc.org

OpenACC

- Implementations:
  - PGI
  - GNU (experimental, >= 5.1)
- Main difference wrt OpenMP
  - scalars are firstprivate by default
  - more concise
  - data handling slightly different

OpenACC

- PROs:
  - High-level. No involvement of OpenCL, CUDA, etc.
  - Single source. No forking off a separate GPU code.
  - Experience shows very favorable comparison to low-level implementations of same algorithms.
  - Performance portable: in principles GPU accelerators and co-processors from any vendor.
  - Incremental. Can be quick.
  - Support AMD gpus (likely)

OpenACC

- CONs:
  - Compilers availability limited (but growing)
  - Not as low level as CUDA or OpenCL

How it looks like

```c
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma acc parallel loop
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

```fortran
subroutine saxpy(n, a, x, y)
  real :: x(:), y(:), a
  integer :: n, i
$!acc parallel loop
  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo
$!acc end parallel loop
end subroutine saxpy

...
! Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
```

Directive Syntax

- C

```
#pragma acc directive [clause [,] clause] …]
```

Often followed by a structured code block

- Fortran

```
!$acc directive [clause [,] clause] …]
```

Often paired with a matching end directive surrounding a structured code block

```
!$acc end directive
```

OpenACC parallel

- Programmer identifies a block of code suitable for parallelization and **guarantees** that no dependency occurs across iterations
- Compiler generates parallel instructions for that loop e.g., a parallel CUDA kernel for a GPU

```
#pragma acc parallel loop
for (int j=0;j<n;j++) {
  for (int i=0;i<n;i++) {
    A[j][i] = B[j][i] + C[j][i]
  }
}
```

OpenACC kernels

- The kernels construct expresses that a region may contain parallelism and the compiler determines what can be safely parallelized.

```
!$acc kernels
do i=1,n
  a(i) = 0.0
  b(i) = 1.0
  c(i) = 2.0
end do
do i=1,n
  a(i) = b(i) + c(i)
end do
!$acc end kernels
```

OpenACC parallel vs kernel

`parallel`

- Requires analysis by programmer to ensure safe parallelism
- Straightforward path from OpenMP
- Mandatory to fully control the different levels of parallelism
- Implicit barrier at the end of the parallel region

`kernels`

- Compiler performs parallel analysis and parallelizes what it believes safe
- Can cover larger area of code with a single directive
- Needs clean codes and sometime directives to help the compiler
- Implicit barrier at the end and between each kernel (e.g. loop)

OpenACC loop

- Applies to a loop which must immediately follow this directive
- Describes:
  - type of parallelism
  - loop-private variables, arrays, and reduction operations
- We already encountered it combined with the parallel directive

```
C
#pragma acc loop [clause …]
{ for block }
```

```
Fortran
!$acc loop [clause …]
{ do block }
```

OpenACC independent

- In a kernels construct, the independent *loop* clause helps the compiler in guaranteeing that the iterations of the loop are independent wrt each other
- E.g., consider m>n

```
#pragma acc kernels
#pragma acc loop independent
for(int i;i<n;i++)
   c[i] = 2.*c[m+i];
```

- In parallel construct the independent clause is implied on all loop directives without a *seq* clause

OpenACC seq

- The *seq* clause specifies that the associated loops have to be **executed sequentially on the accelerator**
- Beware: the loop directive applies to the immediately following loop

```
#pragma acc parallel
#pragma acc loop // independent is automatically enforced
for(int i;i<n;i++)
  for(int k;k<n;k++)
#pragma acc loop seq
    for(int j;j<n;j++)
       c[i][j][k] = 2.*c[i][j+1][k];
```

OpenACC reduction

- The reduction clause on a loop specifies a reduction operator on one or more scalar variables
  - For each variable, a private copy is created for each thread executing the associated loops
  - At the end of the loop, the values for each thread are combined using the reduction clause
- Common operators are supported: + * max min && || ....

```
#pragma acc parallel loop reduction(max:err) shared(A,Anew,m,n)
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]);
  }
}
```

OpenMP 4.5

- Spec. available since Nov. 2015
- Already implemented in:
  - GCC 6.0 (almost)
  - Clang
- Similar directives but:
  - No *independent* clause
  - No *kernels,* you have to be the paranoid!

OpenMP 4.5

• target	• parallel / kernels
• teams	• parallel / kernels
• distribute	• loop gang
• parallel	• parallel / kernels
• for / do	• loop worker or loop gang
• simd	• loop vector
• is_device_ptr(...)	• deviceptr(...)

PGAS model

- Partitioned Global Address Space (PGAS) programming model
- Assumes a global memory address space that is logically partitioned and a portion of it is local to each process or thread.
- A process can directly access a memory portion owned by another process.
- Combine the advantages of a SPMD programming style for distributed memory systems (as employed by MPI) with the data referencing semantics of shared memory systems.

PGAS Model

- Unified Parallel C (UPC)
- CoArray Fortran (CAF)
- X10 (IBM)
- Chapel (CRAY, chapel.cray.com)

PGAS model

Memory models

	Thread Count	Memory Count	Nonlocal Access
Serial	1	1	N/A
OpenMP	1 to p	1	N/A
MPI	p	p	No. Use messages.
UPC, CAF	p	p	YES
X10, Chapel	p	q	YES

Coarray

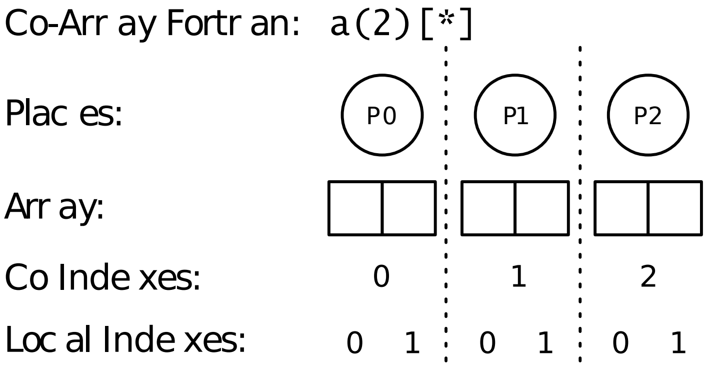
- Cray Compiler (Gold standard - Commercial)
- Intel Compiler (Commercial)
- GNU Fortran (Free - GCC)
- Rice Compiler (Free - Rice University)
- OpenUH (Free - University of Houston)

Unified Parallel C

- extension of the C Programming language designed for high performance computing on large-scale parallel machines
- Same concept of CAF
- Cray compilers, as well as through Berkeley UPC

CAF – basic rules

- A program is treated as if it were replicated at the start of execution (SPMD), each replication is called an image.
- Each image executes asynchronously.
- An image has an image index, that is a number between one and the number of images, inclusive.
- A coarray is indicated by trailing [ ].



Co-Array Fortran: a(2)[*]

Places: P0 P1 P2

Array:

Co Indexes: 0 1 2

Local Indexes: 0 1 0 1 0 1

CAF – basic rules

- A coarray could be a scalar or array, static or dynamic, and of intrinsic or derived type.
- A data object **declared without trailing [ ]** is local.
- If not specified, coarrays on local image are accessed.
- Explicit synchronization statements are used to maintain program correctness

CAF – basic rules

- When we declare a coarray variable the following statements are true:
  - The coarray variable exists on each image.
  - The coarray name is the same on each image.
  - The size is the same on each image.

```
x(:) = y(:)[q]
```

CAF memory decalration

```
!  Scalar  coarray
integer  :: x[*]

! Array  coarray
real , dimension(n) :: a[*]

!  Another  array  declaration
real , dimension(n), codimension [*] :: a

!  Scalar  coarray  corank  3
integer  :: cx[10 ,10, *]

! Array  coarray  corank  3

!  different  cobounds
real :: c(m,n) :: [0:10 ,10 ,*]

!  Allocatable  coarray
real , allocatable  :: mat (: ,:)[:]
allocate(mat(m,n)[*])

!  Derived  type  scalar  coarray
type(mytype) :: xc[*]
```

CAF segments

- A segment is a piece of code between synchronization points. Sync are *SYNC ALL, SYNC MEMORY, SYNC IMAGES*
- The compiler is free to apply optimizations within a segment.
- Segments are ordered by synchronization statement and automatic sync happens at dynamic memory actions ([de]allocate).

```
real :: p[*]                   ! :
                               ! Segment  1
sync  all
if (this_image ()==1)  then    ! Segment  2
  read  (*,*) p                ! :
  do i = 2, num_images ()      ! :
    p[i] = p                   ! :
  end do                       ! :
end if                         ! Segment  2
sync  all
                               ! Segment  3
```

CAF segments

- A segment is a piece of code between synchronization points. Sync are *SYNC ALL, SYNC MEMORY, SYNC IMAGES*
- The compiler is free to apply optimizations within a segment.
- Segments are ordered by synchronization statement and automatic sync happens at dynamic memory actions ([de]allocate).

```
real :: p[*]                    ! :
                                ! Segment  1
sync  all
if (this_image ()==1)  then     ! Segment  2
  read  (*,*) p                 ! :
  do i = 2, num_images ()       ! :
    p[i] = p                    ! :
  end do                        ! :
end if                          ! Segment  2
sync  all
                                ! Segment  3
```

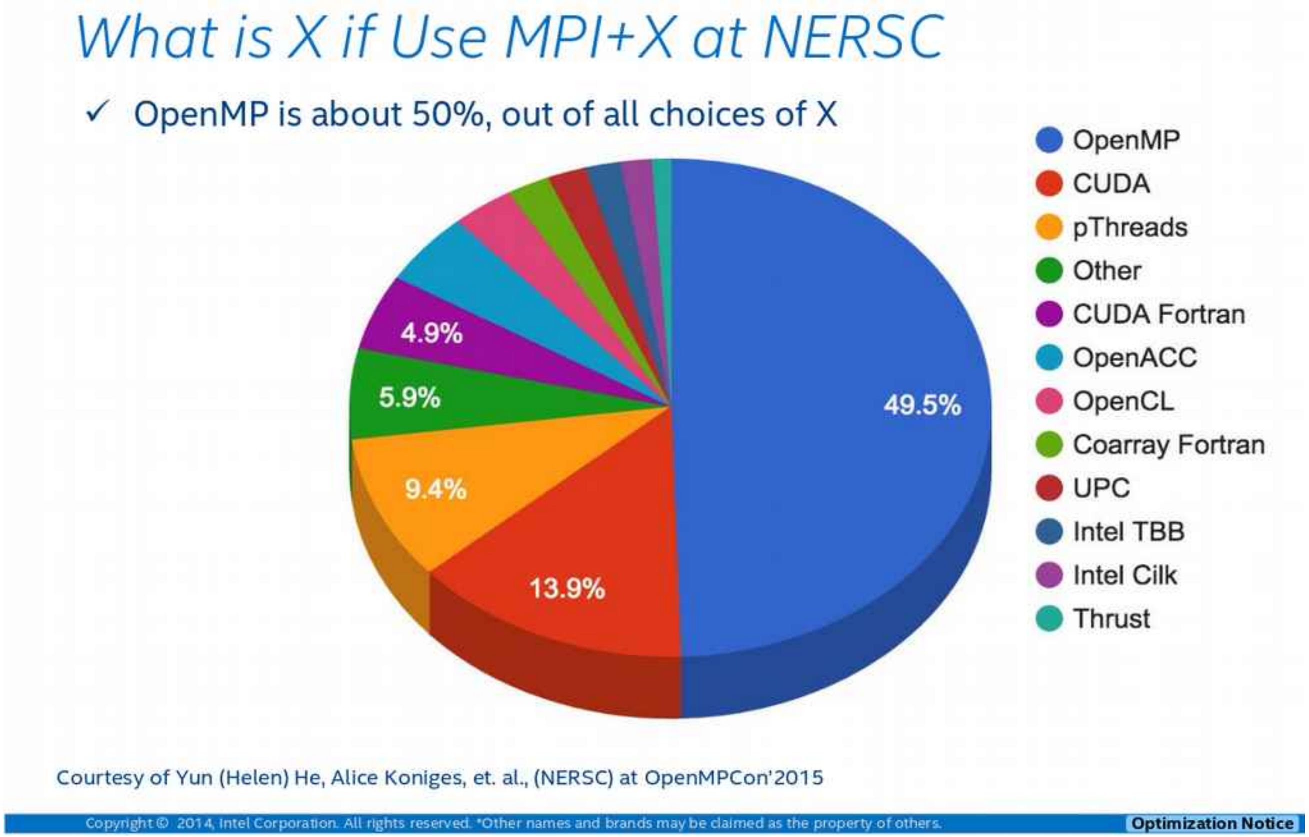
OpenCL

- Similar to CUDA, but even more low level
- Targets all kind of accelerators!
- If you have CUDA kernels, you may get OpenCL kernels rather easily
- Experience: performance not as good as CUDA
- Missing Fortran direct access (C wrap needed)
- Lot of code available, reuse possible.
- **Open standard** maintained by a non-profit technology consortium (Khronos Group).

ROCm

- Alternative to CUDA for (multi) GPU programming
- Extremely new (started in 2016)
  - HC C++ API: C++ 11/14 compiler
  - HIP: Tools and API to convert CUDA to portable C++ API.

Let's take a breath and look around...



What is X if Use MPI+X at NERSC

- ✓ OpenMP is about 50%, out of all choices of X

Courtesy of Yun (Helen) He, Alice Koniges, et. al., (NERSC) at OpenMPCon'2015

Optimization Notice

http://llvm-hpc2-workshop.github.io/slides/Tian.pdf

Take home message

- Complexity will increase
  - Many GPUs
  - Many different *many* Core chips
  - FPGA (?)

Take home message

- Accelerators' architecture is evolving quickly
  - Eg. AMD heavily targeting GPGPU for scientific computation lately.
- **Portability** is recommended for today and for tomorrow
  - Code maintenance
  - Code evolution
- **Open standards** (eg. OpenACC, OpenMP) must be considered.
  - Code not bound to a specific company's will/fate.
  - Community effort for standardization, evolution and support.
- Choose a reasonable compromise between readability, **maintainability** and **performance**.

Take home message

- Use **libraries**
- Separate computation intensive part from the main code: separate independent components. (Eg. Hamiltonian construction & solution)
- Think of **data distribution/locality**
- Facilitate overlap of communication and computation

Take home message