Introduction to
High-Performance Computing

Giorgio Amati
Alessandro Ceci

Corso di dottorato in Ingegneria Aeronautica e Spaziale 2025

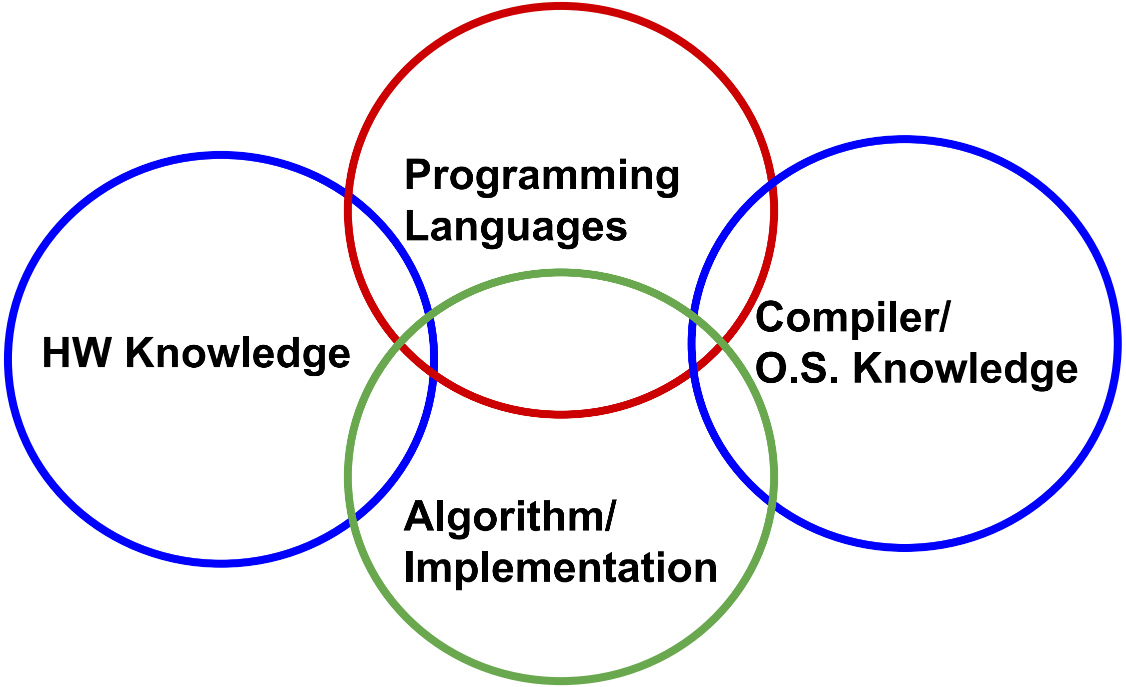
g.amati@cineca.it/g.amaticode@gmail.com
alessandro.ceci@uniroma1.it

Agenda

- ✓ HPC: What is it?
- ✓ Hardware: how it works
- ✓ Algorithm vs. Implementation
- ✓ Compiler + Floating point + I/O
- ✓ HW & Parallel Paradigm
  - ■ Shared Memory parallelization: openmp et al.
  - ■ GPU programming
- ✓ Conclusions & Comments

HPC: what it is?

✓ These are the main skills for efficient HPC



Programming Languages

HW Knowledge

Compiler/ O.S. Knowledge

Algorithm/ Implementation

GPU vs. CPU

- ✓ General-purpose computing on graphics processing units (GPGPU) is the use of a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU). The use of multiple video cards in one computer, or large numbers of graphics chips, further parallelizes the already parallel nature of graphics processing.

- ✓ In brief.
  - GPUs are less “flexible” than CPU
  - GPUs could be much more performing than CPUs
  - Classical example:
    - GPU → BUS
    - CPU → sport car

Why GPUs?

✓ Pro

- GPUs are more powerful: 1 GPU ~ 10x CPUs (Peak Mflops)
- GPUs ask for less space: for the same performance CPUs ask for **~3x racks**
- GPUs are less expensive: for the same peak performance CPUs are **~2x expensive**
- GPUs ask for (relative) less power: for the same peak performance CPUs **~4x energy**

✓ Cons

- GPUs are less flexible than CPUs
- Some algorithms are not GPU-friendly
- There's no common programming model between different vendors
- Porting to GPU is an expensive and error-prone procedure

Different Level of optimization

- ✓ **Single core optimization**
  - ○ Vectorization
  - ○ Data access
  - ○ Serial Optimized libraries
  - ○ Compiler optimization
- ✓ **Single-node optimizations**
  - ○ Intra-node optimization
  - ○ Data access
  - ○ Shared memory Parallelization/Distributed memory Parallelization
  - ○ **Offloading**
  - ○ Shared Memory Optimized Libraries
- ✓ **Multi-node optimizations**
  - ○ Distributed memory optimization (i.e. load balancing)
  - ○ Parallel Optimized libraries
- ✓ **I/O issues**

You cannot skip one single level of optimization!!

Heterogeneous Machine/1

- ✓ Real world systems are “heterogenous”.
- ✓ Up to three different level of parallelization to manage
- ✓ Usually they are a cluster (i.e. distributed memory system) of nodes
    - ○ Each of them is a shared memory system of cores (up to 128 or more)
        - ○ Eventually with GPUs (up to 8)

For example Leonardo has

- ✓ 3456 Nodes, interconnected via an infiniband network at 200Gb/s
    - ○ Each node has 32 Core with 512GB Ram
        - ○ Each node has 4 GPU with 64GB HBM

How to handle this “complexity”???

Heterogeneous Machine/2

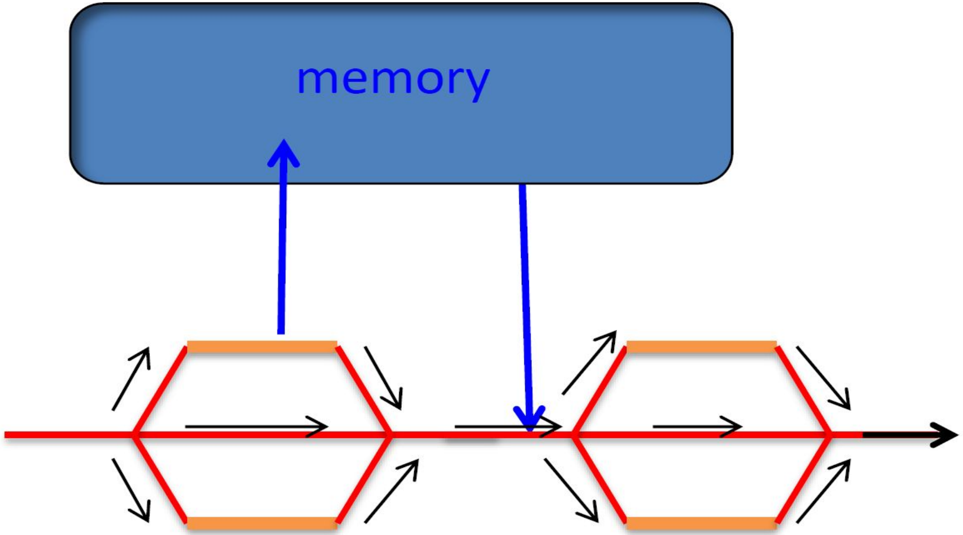
How to handle this “complexity”???

✓ Shared Memory parallelism
  ○ OpenMP (CPU)
  ○ OpenACC (GPU)
  ○ OpenMP Offload (GPU)
  ○ Cuda/Cuda Fortran/HIP (GPU)
  ○ …

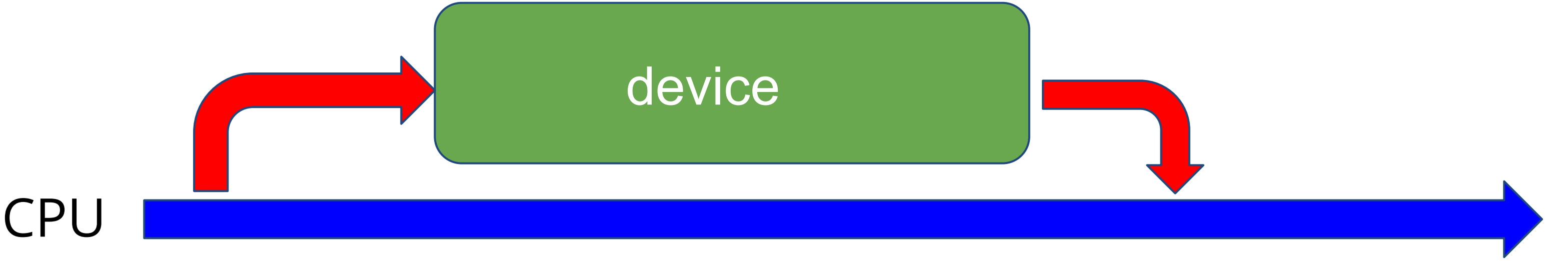
✓ Distributed memory parallelism
  ○ MPI
  ○ …

Shared memory parallelism

- ✓ Work-sharing model
- ✓ One process (master thread) that:
  - ○ “forks” in different threads
  - ○ each thread accesses to a common memory
  - ○ each thread performs some operations
  - ○ all threads join the master threads
- ✓ Risk of “race conditions”
- ✓ No risk of “deadlocks”



Device Offloading



Offloading:

- ✓ Some work is “demanded” to an “external” device (GPU,FPGA)
- ✓ explicit data movement back and from the device
- ✓ the bottleneck is the data movement
- ✓ usually devices has less memory then CPU (Leonardo: 4x64GB vs. 512 GB)

It isn’t a real shared memory parallelism

Device-Host Bandwidth

Host-device model

- Host: CPU and its memory
- Device: GPU (or other) and its memory

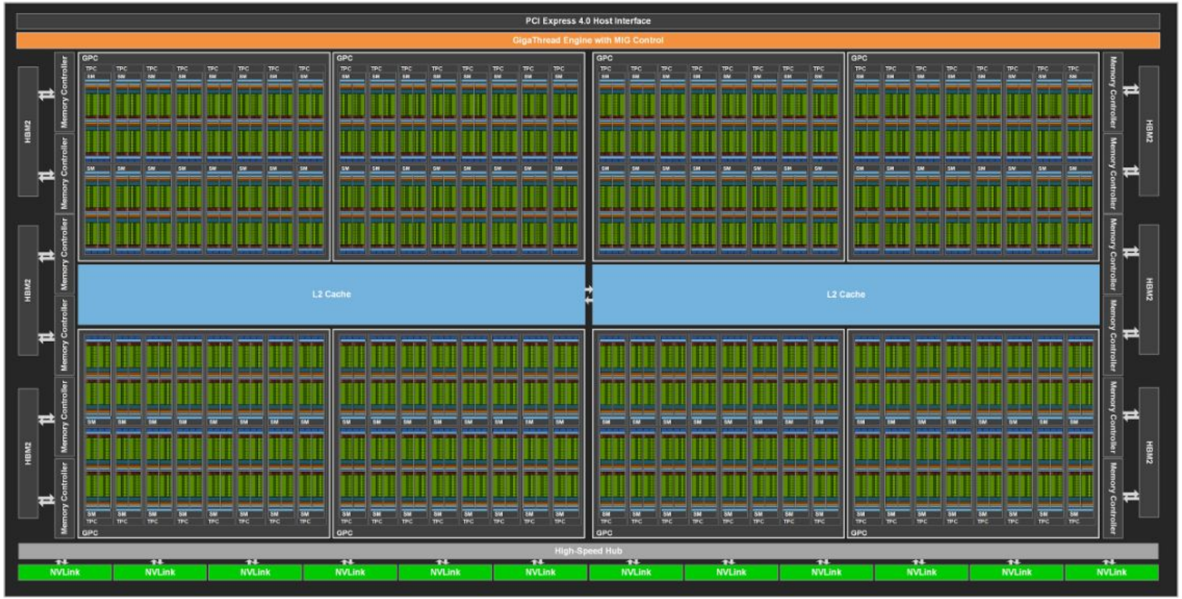
1. Allocate memory on host
2. Data transfer from host to device
3. Execute on device
4. Data transfer from device to host
5. ….

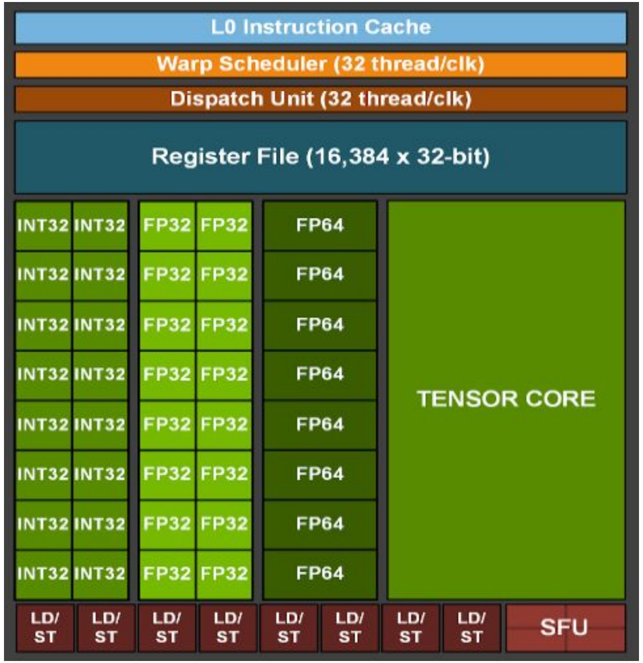
- ✓ CPU = O(1) TF
- ✓ GPU = O(10) TF
- ✓ BW CPU = O(100) GB/s
- ✓ BW GPU = O(1000) GB/s
- ✓ PCIe = 12-48 GB/s
- ✓ NVLINK = 25-400 GB/s

Caveat

- ✓ Focused on NVIDIA GPUs
- ✓ Other GPU present “more or less” similar structure but are different
- ✓ Also keywords present a bias towards NVIDIA

GPU




L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ ST
SFU

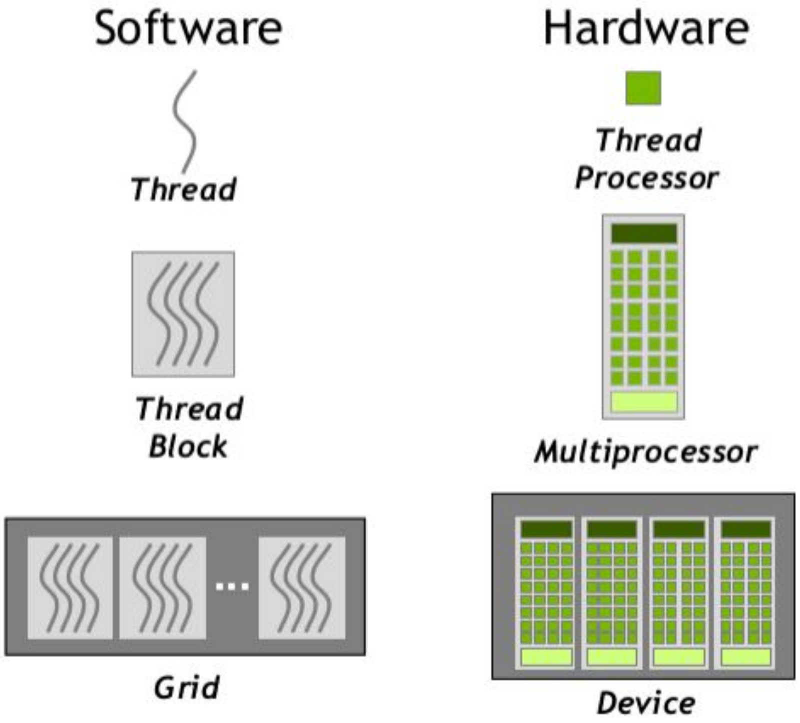
NVIDIA GA100

- ✓ up to 128 Streaming multiprocessor (SM)
- ✓ Each SM has
  - 64 FPU@32bit
  - 32 FPU@64bit
  - 64 INT@32bit

Execution model

- ✓ Heach thread (software) is executed by a thread processor
- ✓ A block of threads (SW) is executed on a multiprocessor (HW)
- ✓ More blocks can share a multiprocessor
- ✓ A grid of blocks (SW) is launched on a device (many multiprocessor, HW)
- ✓ a kernel is “offloaded” to GPU.

A kernel, to be efficient, must generate many many threads!!!



Built-in variables

- ✓ To make “light threads” some built-in variables are defined for GPU to help “work sharing” (type **`dim3`**)
  - ○ Grid dimension: **`gridDim`**
  - ○ Block dimension: **`blockDim`**
  - ○ Block indices: **`blockIdx`**
  - ○ Thread indices: **`threadIdx`**

- ✓ With these variables we can “easily” dispatch threads to threads processors
- ✓ here threads are logically different from openmp threads

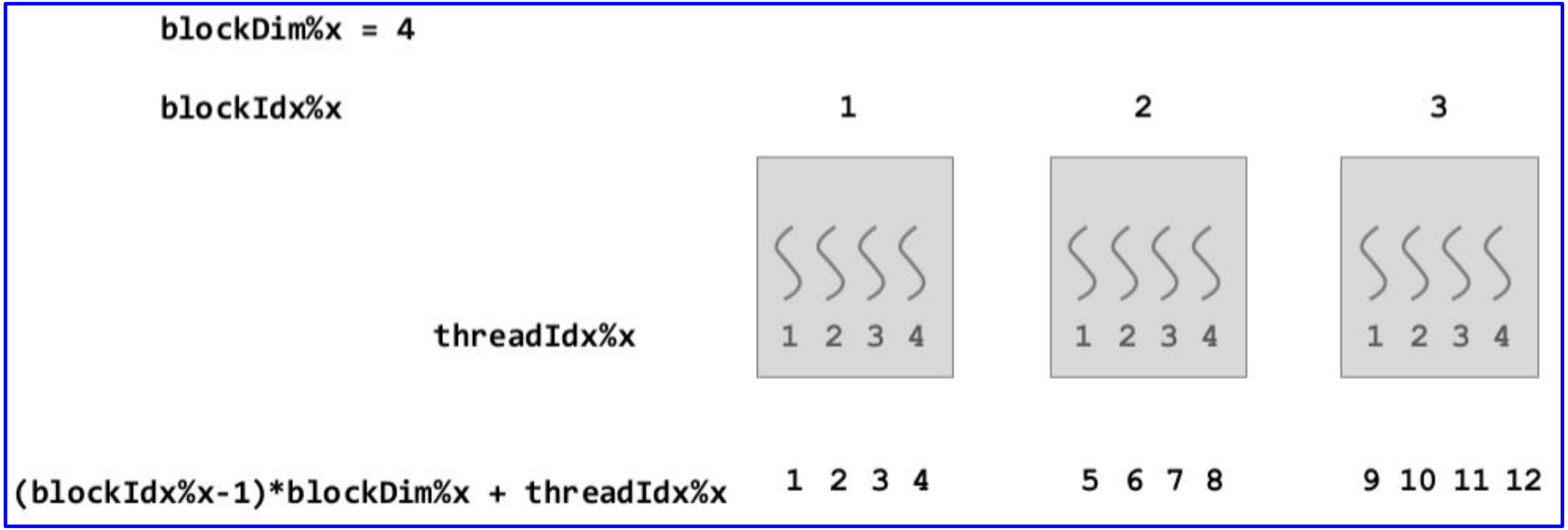
Built-in variable (1D)

✓ 12 threads (SW) → GridDim

✓ 4 thread per Block→ blockDim

✓ 3 Blocks →blockIdx

✓ Single thread id (per block) → threadIdx



blockDim%x = 4

blockIdx%x 1 2 3

threadIdx%x 1 2 3 4 1 2 3 4 1 2 3 4

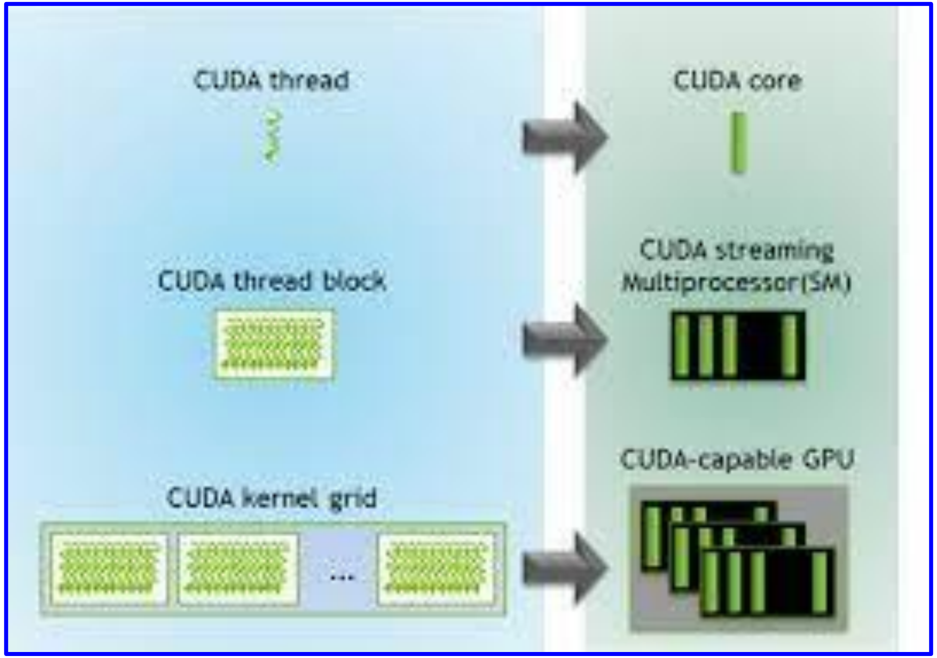
(blockIdx%x-1)*blockDim%x + threadIdx%x 1 2 3 4 5 6 7 8 9 10 11 12

CUDA in few slides

- ✓ **C**ommon **U**nified **D**evice **A**rchitecture
- ✓ Nvidia Proprietary language
- ✓ Offload parallelism: work is demanded to an external accelerator
- ✓ C-based language with extension (Python,Fortran,...)
- ✓ Kernel based parallelism
- ✓ Heavy code modification
- ✓ Very fine control
- ✓ **HIP**: AMD version very similar philosophy

CUDA in few slides

- ✓ Kernel: a function which is flagged to be run on a GPU.
- ✓ A kernel is executed on the core of a multiprocessor inside a thread
- ✓ **A thread can be thought of as just an index $j \in N$, an index of cores in multiprocessors**
- ✓ At any given time, a block of threads is executed on a multiprocessor.



CUDA Example/1

```
* cudaArray* cu_array;
texture<float, 2> tex;
// Allocate array
cudaChannelFormatDesc description = cudaCreateChannelDesc<float>();
cudaMallocArray(&cu_array, &description, width, height);
// Copy image data to array
cudaMemcpy(cu_array, image, width*height*sizeof(float),
cudaMemcpyHostToDevice);
// Bind the array to the texture
cudaBindTextureToArray(tex, cu_array);
// Run kernel
dim3 blockDim(16, 16, 1);
dim3 gridDim(width / blockDim.x, height / blockDim.y, 1);
kernel<<< gridDim, blockDim, 0 >>>(d_odata, height, width);
cudaUnbindTexture(tex);
```

CUDA example/2

```
__global__ void kernel(float* odata, int height, int width)
{
   unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
   unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
   float c = tex2D(tex, x, y);
   odata[y*width+x] = c;
}
....
```

CUDA Fortran Example/1

```
real(my_kind), dimension(:,:), allocatable:: a ! matrix (origin)
real(my_kind), dimension(:,:), allocatable:: b ! matrix (origin)
real(my_kind), dimension(:,:), allocatable:: c ! matrix
real(my_kind), dimension(:,:), device, allocatable:: a_gpu ! matrix (origin)
real(my_kind), dimension(:,:), device, allocatable:: b_gpu ! matrix (origin)
real(my_kind), dimension(:,:), device, allocatable:: c_gpu ! matrix

a_gpu = a
b_gpu = b
c_gpu = c
!$cuf kernel  do <<<*,*>>>
do j = 1, n
    do k = 1, n
        do i = 1, n
            c_gpu(i,j) = c_gpu(i,j) + a_gpu(i,k)*b_gpu(k,j)
        enddo
    enddo
enddo
```

OpenACC in few slides

- ✓ Offload parallelism: work is demanded to an external accelerator
- ✓ **Similar philosophy to OpenMP but**
  - ○ **Implicit data movement back/from GPU, via PCI bus or “ad hoc” link (i.e. NVlink)**
  - ○ **More sensible to data movement, this is the bottleneck for many codes**
- ✓ Programmer has to take care of data movement back/from GPU and which piece of work offload to gpu
- ✓ Incremental (loop by loop) but **not in performance (data movement hides performance)**
- ✓ **Directive based**: no code modification (in principle)
- ✓ Performance heavily dependent from the compiler (and release) used
  - ○ Not supported by all compilers (Nvidia, CRAY-AMD, IBM, gnu)

OpenACC in few slides: C

✓ the **`kernels`** directive is used to instruct the compiler to **analyze and parallelize** the enclosed code region automatically.

✓ Compiler do all the work

✓ Matrix matrix multiplication:

```
#pragma acc kernels
for (i = 0; i < nn; i++) {
    for (k = 0; k < nn; k++) {
            for (j = 0; j < nn; j++) {
                c[i][j] = c[i][j] + a[i][k]*b[k][j];
            }
        }
}
```

OpenACC in few slides: Fortran

✓ The **`parallel`** directive is used to **explicitly define a region of code** that should be executed in parallel on an accelerator, such as a GPU.

✓ With this directive programmer has more control

✓ Matrix matrix multiplication:

```
!$acc parallel copyin(a,b) copyout(c)
!$acc loop parallel
  do j=1, n
     do k=1, n
        do i=1, n
           c(i,j) = c(i,j) + a(i,k)*b(k,j)
        end do
     end do
  end do
!$acc end parallel
```

OpenACC data movement directives

To directly handle data movement
Data movement directives

✓ **`data/end data:`** open/close a data region
✓ **`copy(list)`**: from CPU to GPU and back
✓ **`copyin(list)`**: from CPU to GPU (e.g. initialization)
✓ **`copyout(list)`**:from GPU to CPU (e.g. I/O operation)
✓ **`update(list)`**
- ○ **`host(list)`**: update host (from gpu)
- ○ **`device(list)`**: update device (from cpu)

✓ …

OpenACC in few slides:Memory Management

- ✓ In OpenACC, memory management refers to how data is transferred and synchronized between the host (CPU) and the device (GPU or accelerator). Since the host and device have separate memory spaces, OpenACC provides directives to manage this explicitly or automatically, if supported by compiler/HW
- ✓ In a **`data`** region all variables defined are present in the GPU HMB Memory, no need to copy back in RAM

```
!$acc data copyin(a01,.....a19)
      call work1(a01,....a19)
      call work2(a01,....a19)
      call work3(a01,....a19)
         ….
!$acc end data
```

- ✓ Take care oh I/O (you have to write back the data to CPU!!!!)

OpenACC in few slides:Errors!

```
!$acc data copy(a01,.....a18)
      call work1(a01,....a19)
      call work2(a01,....a19)
      call work3(a01,....a19)
         ….
!$acc end data
```

```
!$acc data copy(a01,.....a18)
      …
      b01=a01
      …
      a01=b01
      …
!$acc end data
```

OpenACC in few slides: Compiling….

```
nvfortran -acc -Minfo=acc -ta=tesla:cc80,managed
```

- ✓ **`-acc:`** activate openacc
- ✓ **`-Minfo=acc:`** info about transformation (standard output)
- ✓ **`-ta:`** target acrticecture
  - ○ **`cc??:`** compute capability (i.e. GPU)
  - ○ **`managed`**: automatic memory management

OpenACC in few slides: Compiling….

```
nvfortran -acc -Minfo=acc -ta=tesla:cc80,managed -Mcontiguous -fast -Mcontiguous
-Mnodepchk  -DSERIAL -DPGI -DOPENACC -DDRIVEN  -DPGI -DSENDRECV   -c col_MC.F90
nvfortran-Warning-The flag -ta has been deprecated, please use -acc and -gpu
instead.
col_mc:
    111, Generating implicit firstprivate(l,n,m)
         Generating NVIDIA GPU code
    113, !$acc loop gang collapse(2) ! blockidx%x
    114,   ! blockidx%x collapsed
    118, !$acc loop vector(128) ! threadidx%x
    111, Generating implicit copyin(a01(0:l-1,2:m+1,1:n),...)
         Generating implicit copy(a19(1:l,1:m,1:n))
         Generating implicit copyout(b01(1:l,1:m,1:n),.......)
         [if not already present]
    118, Loop is parallelizable
```

OpenACC (H100)

```
!$acc data copyin(a,b) copy(c)
!$acc kernels
   do j = 1, n
      do k = 1, n
         do i = 1, n
            c(i,j) = c(i,j) + a(i,k)*b(k,j)
         enddo
      enddo
   enddo
!$acc end kernels
!$acc end data
```

```
out.1024.log: GPU:MFLOPS                    7743.039612676056
out.2048.log: GPU:MFLOPS                    77158.71052631579
out.4096.log: GPU:MFLOPS                    451081.6923076923
out.8192.log: GPU:MFLOPS                    1125899.904000000
```

OpenMP Offload in few slides

- ✓ Offload parallelism: work is demanded to an external accelerator
- ✓ Similar philosophy to OpenACC but inserted in OpenMP framework
- ✓ **Same issues for OpenACC: programmer has to take care of data movement back/from GPU and which piece of work offload to gpu**
- ✓ **Incremental (loop by loop) but...**
- ✓ Directive based: no code modification (in principle)
- ✓ Performance heavily dependent from the compiler (and release) used
  - ○ in general slower the OpenACC (to maintain backward compatibility)
  - ○ e.g. pointers are not supported by gnu for release < 12.0

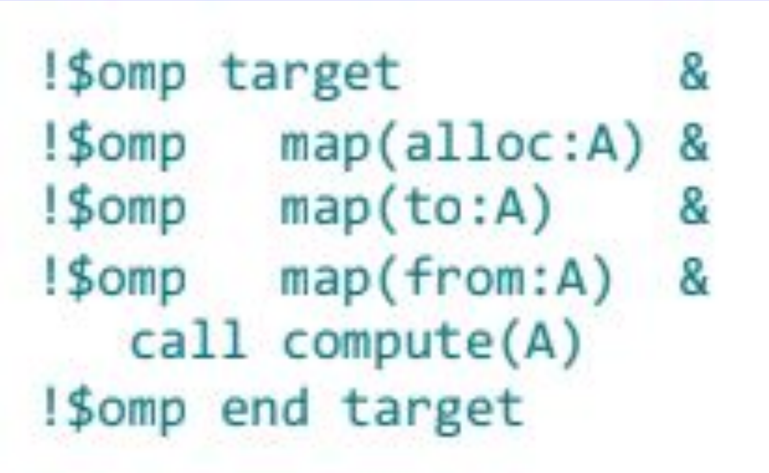
OpenMP Offload main constructs

OpenMP separates offload and parallelism

✓ **`target`** directive transfer control and data from the host to the device

✓ **`map`**:

- ○ **`to`** → from CPU to GPU
- ○ **`from`** → from GPU to CPU
- ○ **`tofrom`** →to/from CPU GPU
- ○ **`alloc`** → allocate to GPU



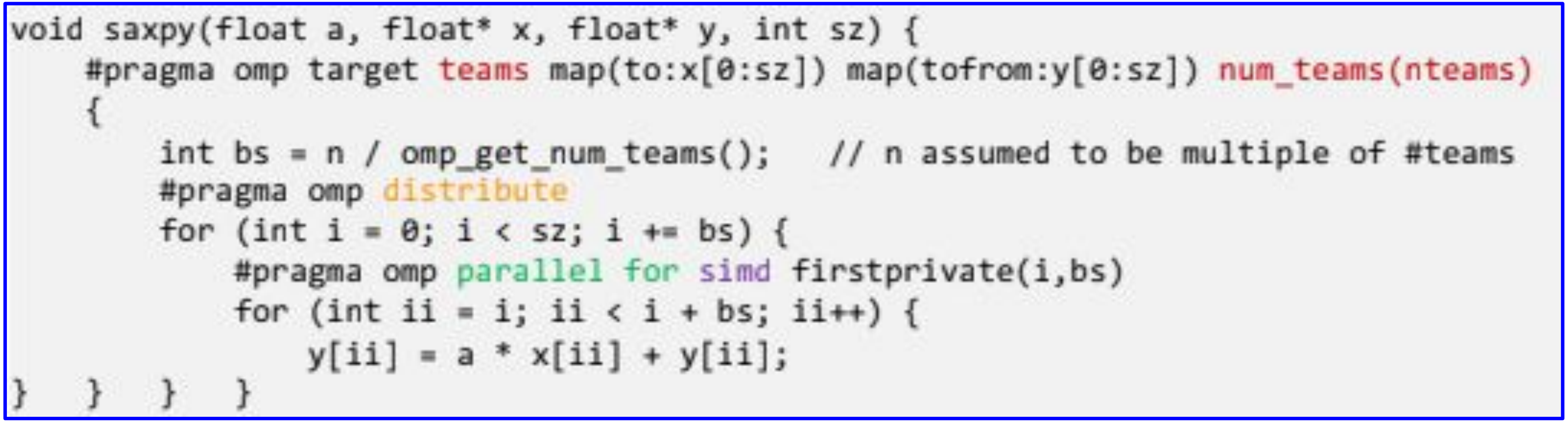
```
!$omp target         &
!$omp   map(alloc:A) &
!$omp   map(to:A)    &
!$omp   map(from:A)  &
   call compute(A)
!$omp end target
```

OpenMP Offload main constructs

OpenMP separates offload and parallelism (hierarchical model)

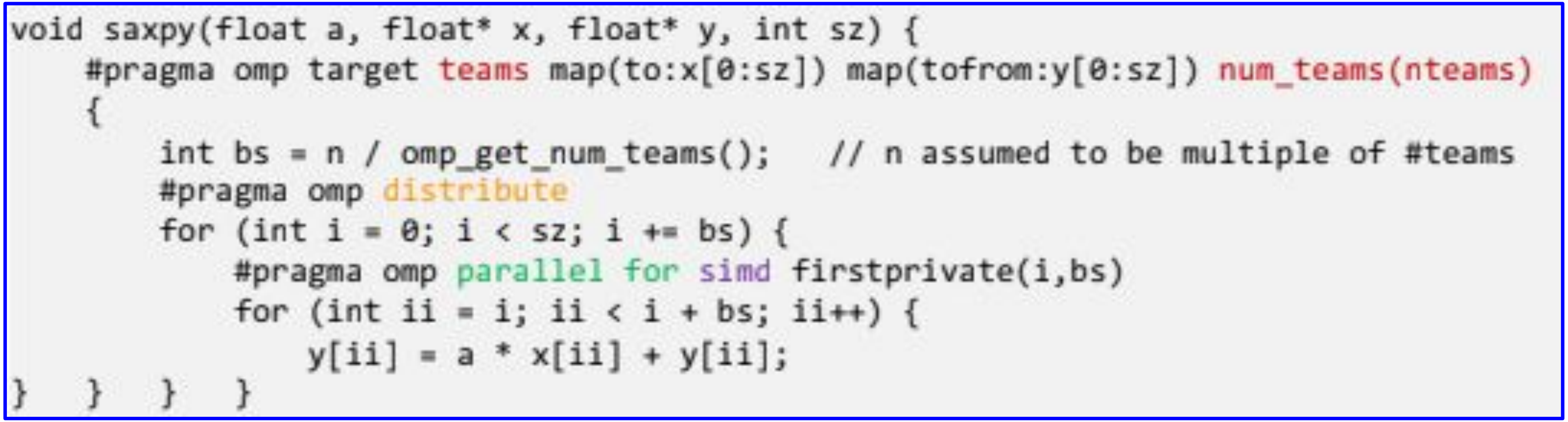
✓ Assign the outer loop to **`teams`**

✓ Assign the inner loop to the **`threads`**

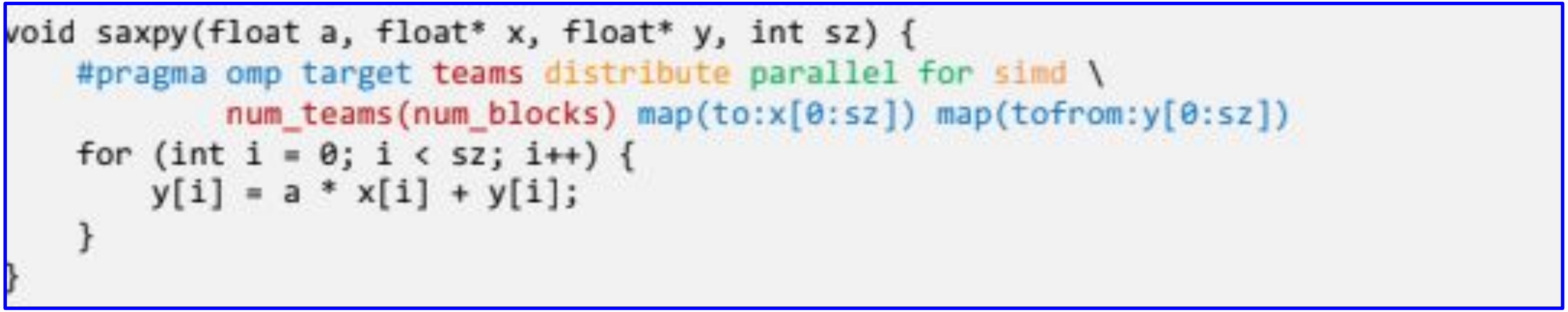


```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target teams map(to:x[0:sz]) map(tofrom:y[0:sz]) num_teams(nteams)
    {
        int bs = n / omp_get_num_teams();   // n assumed to be multiple of #teams
        #pragma omp distribute
        for (int i = 0; i < sz; i += bs) {
            #pragma omp parallel for simd firstprivate(i,bs)
            for (int ii = i; ii < i + bs; ii++) {
                y[ii] = a * x[ii] + y[ii];
}   }   }   }
```

OpenMP Offload main constructs



```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target teams map(to:x[0:sz]) map(tofrom:y[0:sz]) num_teams(nteams)
    {
        int bs = n / omp_get_num_teams();   // n assumed to be multiple of #teams
        #pragma omp distribute
        for (int i = 0; i < sz; i += bs) {
            #pragma omp parallel for simd firstprivate(i,bs)
            for (int ii = i; ii < i + bs; ii++) {
                y[ii] = a * x[ii] + y[ii];
}   }   }   }
```



```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target teams distribute parallel for simd \
            num_teams(num_blocks) map(to:x[0:sz]) map(tofrom:y[0:sz])
    for (int i = 0; i < sz; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

OpenMP Offload in few slides: Compiling….

```
nvfortran -mp=gpu -Minfo=all

timing:
    20, FMA (fused multiply-add) instruction(s) generated
mm:
    58, !$omp parallel
    97, !$omp target teams distribute parallel do
        97, Generating "nvkernel_MAIN__F1L97_3" GPU kernel
    97, Generating map(to:a(:,:))
        Generating map(from:c(:,:))
        Generating map(to:b(:,:))
timing:
    20, FMA (fused multiply-add) instruction(s) generated
```

OpenMP Offload (H100)

```
!$OMP target teams distribute parallel do collapse(2) map(to:a,b) map(from:c)
    do j = 1, n
       do i = 1, n
           do k = 1, n
              c(i,j) = c(i,j) + a(i,k)*b(k,j)
           enddo
       enddo
   enddo
```

```
out.1024.log: GPU:MFLOPS                    7635.497395833333
out.2048.log: GPU:MFLOPS                    73300.77499999999
out.4096.log: GPU:MFLOPS                    359024.2040816327
out.8192.log: GPU:MFLOPS                    679891.2463768116
```

DO CONCURRENT & C++ parallel algorithms

- ✓ Language construct, defined in the standard, that allow to express loop-level parallelism with (possible) back-end to GPU.
- ✓ Parallelization/porting is left to the compiler
- ✓ Not suitable for the best performance but (not even supposed to), if compiler is smart enough, good performance
- ✓ Supported by few compilers (no AMD GPU)

- ✓ "Cross-platform programming model for many-core lattice Boltzmann simulations". Jonas Latt et al, (https://doi.org/10.1371/journal.pone.0250306)
- ✓ https://developer.nvidia.com/blog/accelerating-fortran-do-concurrent-with-gpus-and-the-nvidia-hpc-sdk/

DO Concurrent (H100)

```
do concurrent(j=1:n)
    do k = 1, n
        do i = 1, n
            c(i,j) = c(i,j) + a(i,k)*b(k,j)
        enddo
    enddo
enddo
```

```
out.1024.log: concurrent:MFLOPS                     274877.9062500
out.2048.log: concurrent:MFLOPS                     1099511.625000
out.4096.log: concurrent:MFLOPS                     1759218.600000
out.8192.log: concurrent:MFLOPS                     1695632.385542
```

Do concurrent (H100)

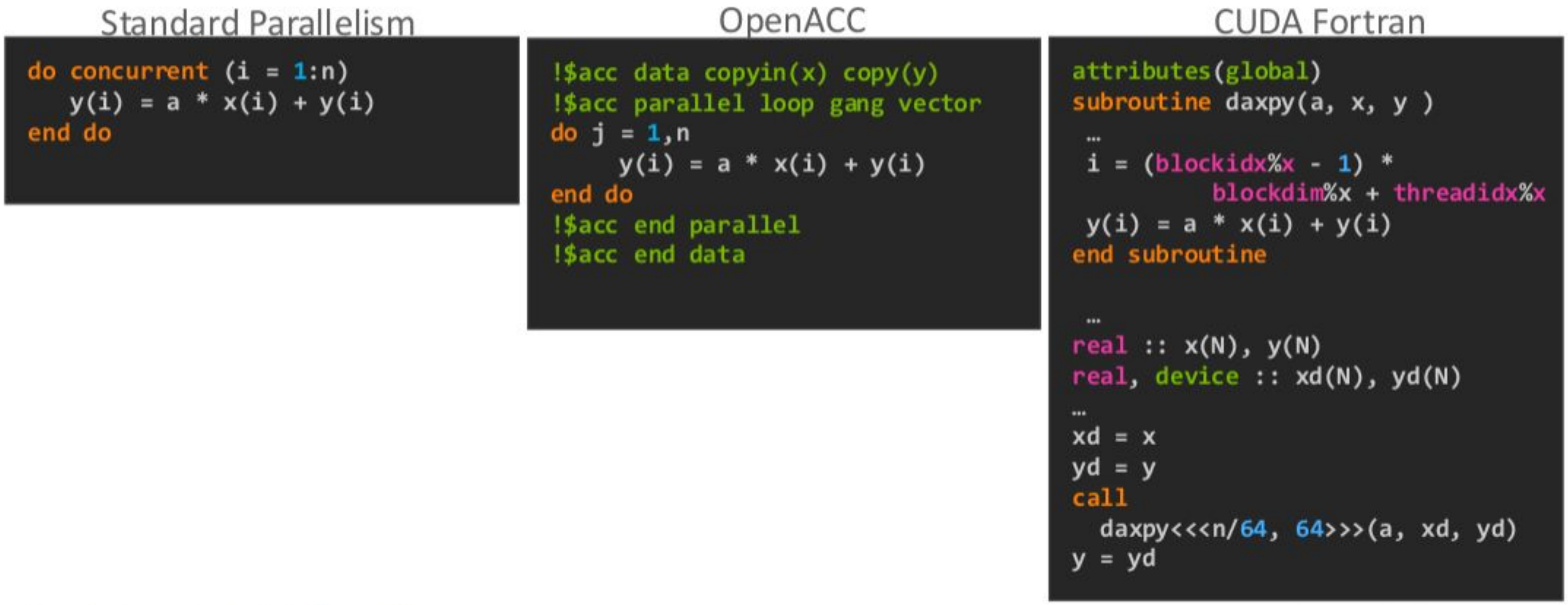
```
do concurrent(j=1:n)
    do k = 1, n
        do i = 1, n
            c(i,j) = c(i,j) + a(i,k)*b(k,j)
        enddo
    enddo
enddo
```

- ✓ SERIAL: `nvfortran -Minfo=all -O3`
- ✓ MULTICORE: `nvfortran -Minfo=all -O3 -stdpar=multicore`
- ✓ GPU: `nvfortran -Minfo=all -O3 -stdpar=gpu`

```
SERIAL, concurrent:MFLOPS                       4617.3716535
MULTICORE,concurrent:MFLOPS                   102280.1511627
GPU,concurrent:MFLOPS                        1954687.3333333
```

Recap….

✓ Possible approach (fortran)



Standard Parallelism

```
do concurrent (i = 1:n)
   y(i) = a * x(i) + y(i)
end do
```

OpenACC

```
!$acc data copyin(x) copy(y)
!$acc parallel loop gang vector
do j = 1,n
     y(i) = a * x(i) + y(i)
end do
!$acc end parallel
!$acc end data
```

CUDA Fortran

```
attributes(global)
subroutine daxpy(a, x, y )
 …
 i = (blockidx%x - 1) *
          blockdim%x + threadidx%x
 y(i) = a * x(i) + y(i)
end subroutine

 …
real :: x(N), y(N)
real, device :: xd(N), yd(N)
…
xd = x
yd = y
call
  daxpy<<<n/64, 64>>>(a, xd, yd)
y = yd
```

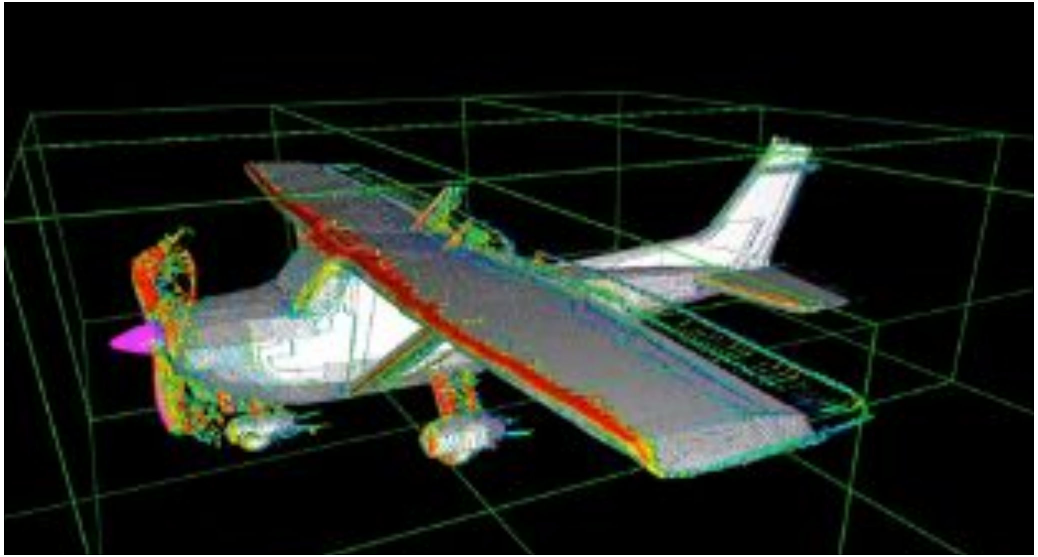
OpenCL & Sycl

- ✓ OpenCL: Open standard for heterogeneous systems (https://www.khronos.org/opencl/)
- ✓ SyCl: cross-platform abstraction layer that enables code for heterogeneous processors to be written using standard ISO C++(https://www.khronos.org/sycl/)
- ✓ It could be a good solution for C++ programmer/code
- ✓ The (nasty) question is how much it is supported by the vendor with respect to their “proprietary” solution

- ✓ Personal comment: It works quite smoothly on AMD MI100 GPU
- ✓ https://www.youtube.com/playlist?app=desktop&list=PLA-vfTt7YHI1SAiiVqviveaSDa8DSt7j9

OpenCL & Sycl

- ✓ Frankenstein-like code
- ✓ Same program using different GPUs (intel,NVIDIA,AMD) at the same time
- ✓ https://www.linkedin.com/feed/update/urn:li:activity:7310377332018454528/



intrinsic function (matmul)

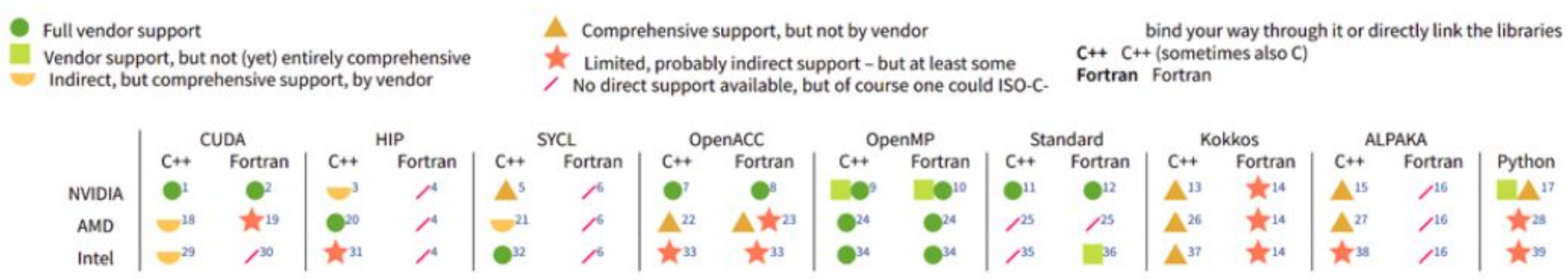
✓ nvfortran -O3 -acc -gpu=managed -cuda -cudalib -Minfo=acc"

```
#ifdef _OPENACC
    use cutensorex
#endif
…
    real(my_kind), dimension(:,:), allocatable:: a ! matrix (origin)
    real(my_kind), dimension(:,:), allocatable:: b ! matrix (origin)
    real(my_kind), dimension(:,:), allocatable:: c ! matrix
…
    c =  matmul(a,b)
```

```
out.1024.log: matmul:MFLOPS                           8329.633522727272
out.2048.log: matmul:MFLOPS                           64677.15441176471
out.4096.log: matmul:MFLOPS                           533096.5454545454
out.8192.log: matmul:MFLOPS                           3803715.891891892
out.16384.log: matmul:MFLOPS                          20849998.22222222
```

Porting to GPU: state of the art

- ✓ Up to now no clear and portable solution
- ✓ No silver bullet!
- ✓ https://github.com/AndiH/gpu-lang-compat



● Full vendor support
🟩 Vendor support, but not (yet) entirely comprehensive
◡ Indirect, but comprehensive support, by vendor
▲ Comprehensive support, but not by vendor
★ Limited, probably indirect support – but at least some
/ No direct support available, but of course one could ISO-C-bind your way through it or directly link the libraries
C++ C++ (sometimes also C)
Fortran Fortran

	CUDA		HIP		SYCL		OpenACC		OpenMP		Standard		Kokkos		ALPAKA		
	C++	Fortran	C++	Fortran	C++	Fortran	C++	Fortran	C++	Fortran	C++	Fortran	C++	Fortran	C++	Fortran	Python
NVIDIA	●[1]	●[2]	◡[3]	/[4]	▲[5]	/[6]	●[7]	●[8]	🟩●[9]	🟩●[10]	●[11]	●[12]	▲[13]	★[14]	▲[15]	/[16]	🟩▲[17]
AMD	◡[18]	★[19]	●[20]	/[4]	◡[21]	/[6]	▲[22]	▲★[23]	●[24]	●[24]	/[25]	/[25]	▲[26]	★[14]	▲[27]	/[16]	★[28]
Intel	◡[29]	/[30]	★[31]	/[4]	●[32]	/[6]	★[33]	★[33]	●[34]	●[34]	/[35]	🟩[36]	▲[37]	★[14]	★[38]	/[16]	★[39]

Performance portability: recap

✓ Up to now no clear and portable solution

A. **Intrinsic language constructs** (i.e., do concurrent Fortran or parallel stl C++)
B. **Vendor specific paradigms** (i.e., CUDA/HIP/OneAPI)
C. **OpenCL standard**
D. **Directive-based standards** (i.e., OpenMP/OpenACX)
E. **high-level portability framework** (i.e., Kokkos/Legion/OpenSYCL/alpaka/RAJA)
F. **External GPU-based subroutine libraries** (i.e., PeTSc)

A --> Not "complete" or even "partial" support by compilers
B --> it should give best performance. Translation between paradigm (e.g., CUDA --> HIP) could be not so simple (e.g., for Fortran)
C --> Code verbose, performance could be hard to achieve, only C.
D --> Not "complete" support and performance could be difficult to achieve
E --> Implies (serious) code rewriting and only some languages are supported (e.g., C/C++)
F --> Only few Algorithm were ported to GPU

Make it harder: OpenMP+OpenACC

- ✓ Each thread controls one GPU

```
!$OMP PARALLEL DEFAULT(NONE) &
!$OMP PRIVATE(i,k,jstart,jstop,idthread) SHARED(a,b,c,n,delta)
     idthread = OMP_GET_THREAD_NUM()
     jstart = 1 + idthread*delta
     jstop =  (idthread+1)*delta
     call acc_set_device_num(idthread, acc_device_nvidia)!
!$acc data copyin(a,b(:,jstart:jstop)) copyin(c(:,jstart:jstop))
!$acc kernels
     do j = jstart, jstop
          do i = 1, n
                   do k = 1, n
                        c(i,j) = c(i,j) + a(i,k)*b(k,j)
                  enddo
          enddo
     enddo
!$acc end kernels
!$acc update host(c(:,jstart:jstop))
!$acc end data
!$OMP END PARALLEL
```

Few References

- ✓ https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial
- ✓ ISC24 Tutorial on OpenMP
    - ○ (some slides & images were “borrowed” from here)