Intro to OpenACC

Lecture 3

Sunita Chandrasekaran
Associate Professor, University of Delaware
PDC Summer School, Aug 2023

Materials also prepared by Dr. Felipe Cabarcas,
Postdoctoral Fellow, UDEL

Methods for Programming GPUs

Kernel-based Languages

- AMD ROCm
- OpenCL™
- NVIDIA CUDA

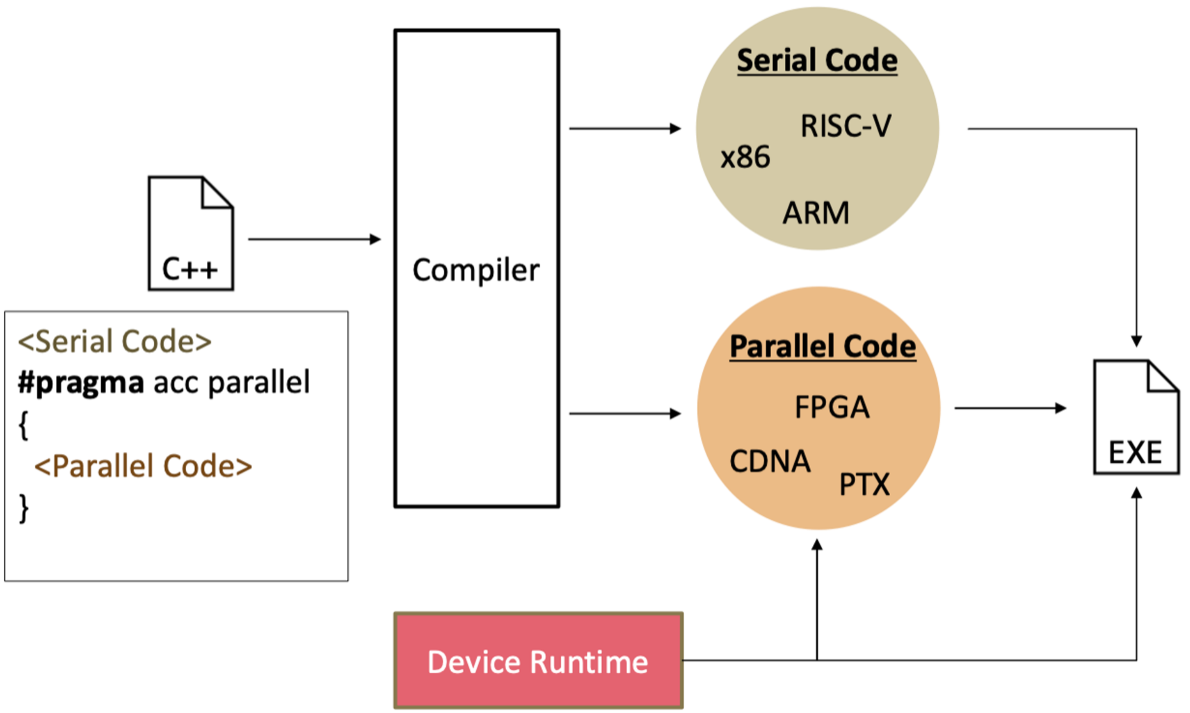
Directive-based/Abstract Programming Models

- OpenMP®
- SYCL™
- OpenACC
- kokkos

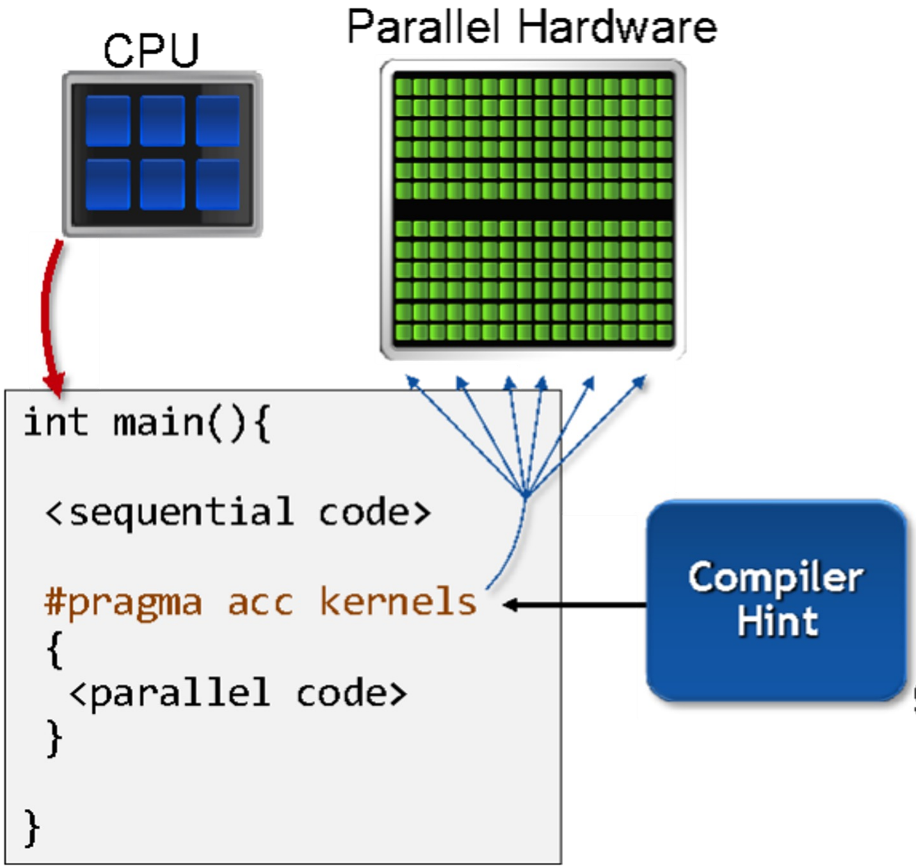
Libraries

- Thrust
- heFFTe
- MAGMA

Directive-Based GPU Programming Models



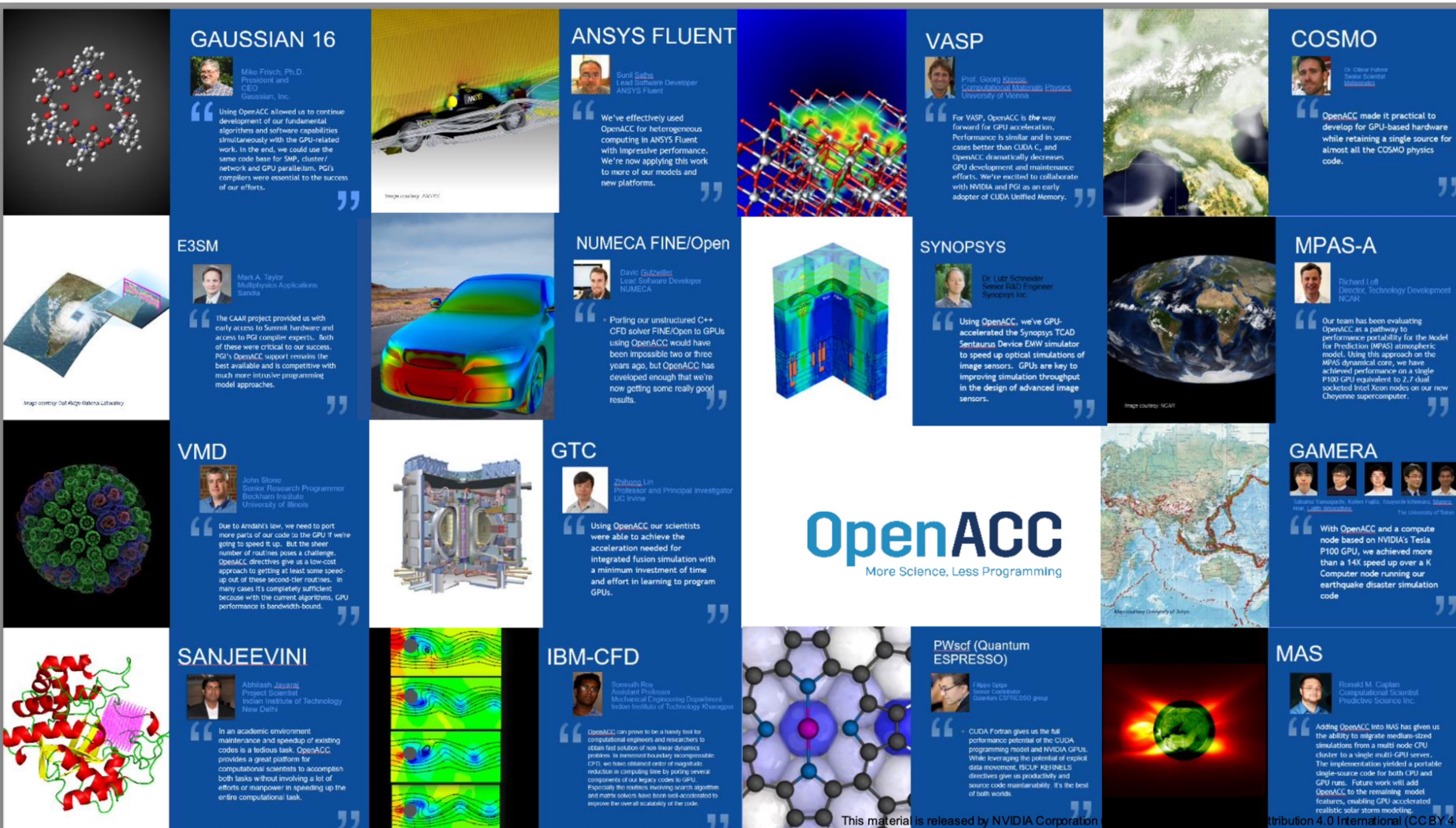
OpenACC functionality



The programmer will give hints to the compiler about which parts of the code to parallelize.

The compiler will then generate parallelism for the target parallel hardware.

University of Delaware



Why OpenACC

Incremental

- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

Single Source

- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine
- Sequential code is maintained

Low Learning Curve

- OpenACC is meant to be easy to use, and easy to learn
- Programmer remains in familiar C, C++, or Fortran
- No reason to learn low-level details of the hardware.

Target Platforms (OpenACC)

- Intel and AMD's x86 (multicore systems)
  - Haswell, Broadwell, Skylake, Icelake
- NVIDIA compilers (nvc) target NVIDIA GPUs
  - All NVIDIA GPUs
- Mentor Graphics compilers (GNU GCC) target both NVIDIA and AMD GPUs (to an extent)
  - AMD Radeon Tahiti (HD 7900), Cape Verde (HD 7700), Spectre (Kaveri APU)
- IBM OpenPOWER 8, 9, 10…

OpenACC syntax

Syntax for using OpenACC directives in code

C/C++

```
#pragma acc directive clauses
<code>
```

Fortran

```
!$acc directive clauses
<code>
```

- A *pragma* in C/C++ gives instructions to the compiler on how to compile the code. Compilers that do not understand a particular pragma can freely ignore it.
- A *directive* in Fortran is a specially formatted comment that likewise instructions the compiler in it compilation of the code and can be freely ignored.
- "*acc*" informs the compiler that what will come is an OpenACC directive
- *Directives* are commands in OpenACC for altering our code.
- *Clauses* are specifiers or additions to directives.

OpenACC parallel loop directive

Parallelizing a single loop

C/C++

```
#pragma acc parallel
{
  #pragma acc loop
  for(int i = 0; j < N; i++)
    a[i] = 0;
}
```

Fortran

```
!$acc parallel
  !$acc loop
  do i = 1, N
    a(i) = 0
  end do
!$acc end parallel
```

- Use a **parallel** directive to mark a region of code where you want parallel execution to occur
- The **loop** directive is used to instruct the compiler to parallelize the iterations of the next loop to run across the parallel gangs

OpenACC parallel loop directive

Parallelizing a single loop

```
#pragma acc parallel loop
for(int i = 0; j < N; i++)
  a[i] = 0;
```

- This pattern is so common that you can do all of this in a single line of code
- In this example, the parallel loop directive applies to the next loop
- This directive both marks the region for parallel execution and distributes the iterations of the loop.
- When applied to a loop with a data dependency, parallel loop may produce incorrect results

OpenACC parallel loop directive

Parallelizing multiple loops

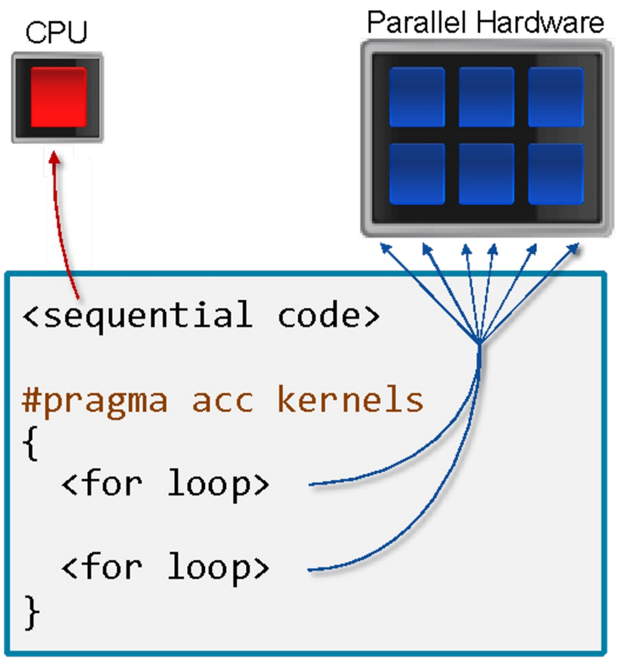
```
#pragma acc parallel loop
for(int i = 0; i < N; i++)
  a[i] = 0;

#pragma acc parallel loop
for(int j = 0; j < M; j++)
  b[j] = 0;
```

- To parallelize multiple loops, each loop should be accompanied by a parallel directive
- Each parallel loop can have different loop boundaries and loop optimizations
- Each parallel loop can be parallelized in a different way
- This is the recommended way to parallelize multiple loops. Attempting to parallelize multiple loops within the same parallel region may give performance issues or unexpected results

OpenACC kernels

Compiler directed parallelization



- The kernels directive instructs the compiler to search for parallel loops in the code
- The compiler will analyze the loops and parallelize those it finds safe and profitable to do so
- The kernels directive can be applied to regions containing multiple loop nests

OpenACC kernels

Parallelizing a single loop

C/C++

```
#pragma acc kernels
for(int i = 0; j < N; i++)
  a[i] = 0;
```

Fortran

```
!$acc kernels
do i = 1, N
  a(i) = 0
end do
!$acc end kernels
```

- In this example, the kernels directive applies to the next for loop
- The compiler will take the loop, and attempt to parallelize **and optimize the loop**
- If the compiler decides that the loop is not parallelizable, it will not parallelize the loop

OpenACC kernels

Parallelizing multiple loops

```c
#pragma acc kernels
{
  for(int i = 0; i < N; i++)
    a[i] = 0;

  for(int j = 0; j < M; j++)
     b[j] = 0;
}
```

```fortran
!$acc kernels
  do i = 1, N
    a(i) = 0
  end do

  do j = 1, M
    b(j) = 0
  end do
!$acc end kernels
```

- In this example, we mark a region of code with the kernels directive
- The compiler will attempt to parallelize all loops within the kernels region
- Each loop can be parallelized/optimized in a different way

Kernels

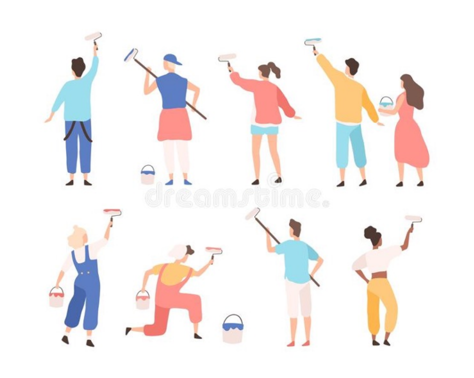
- Compiler decides what to parallelize with direction from user
- Compiler guarantees correctness
- Can cover multiple loop nests

Parallel

- Programmer decides what to parallelize and communicates that to the compiler
- Programmer guarantees correctness
- Must decorate each loop nest

When fully optimized, both will give similar performance.

Three levels of parallelism

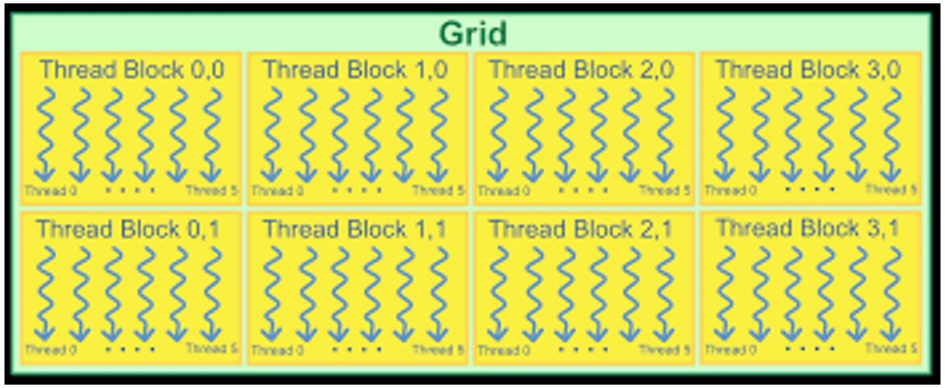


- Gang
  - Like work *crews* they are completely independent of each other and may operate in parallel or even at different times

- Worker
  - Individual *painters* they can operate on their own but may also share resources with other workers in the same gang
- Vector
  - *Paint roller* is the vector
  - where the *width* of the roller represents the vector length.

3 levels of parallelism

- **Gang** OpenACC gang is a threadblock
  - **gang** will apply gang-level parallelism to the loop
- **Worker** OpenACC worker is effectively a warp (a group of threads)
  - **worker** will apply worker parallelism to the loop
- **Vector** OpenACC vector is a CUDA thread
  - **vector** will apply vector-level parallelism to the loop



3 levels of parallelism

- **Gang** OpenACC gang is a threadblock

- **Worker** OpenACC worker is effectively a warp (a group of threads)

Multiple warps

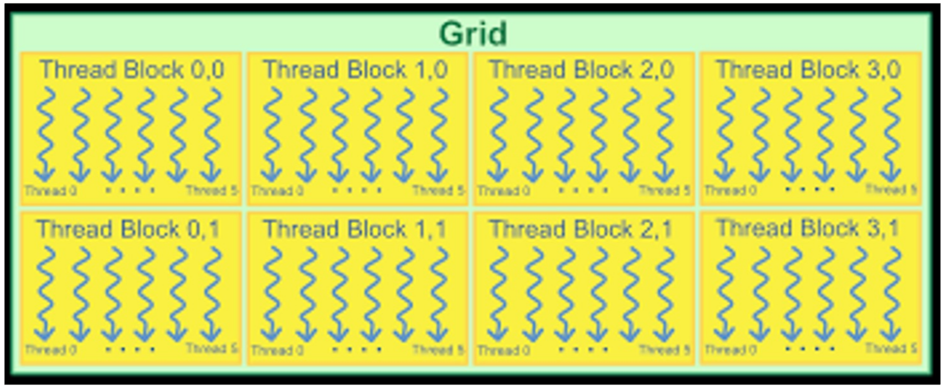
- **Vector** OpenACC vector is a thread

Multiple threads

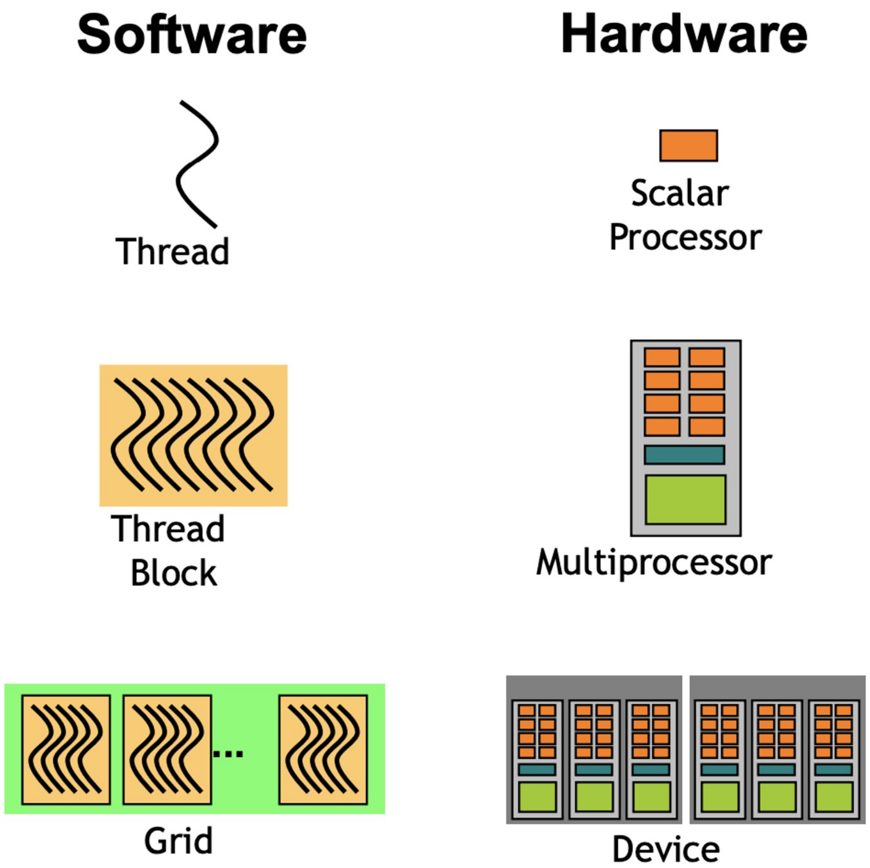
3 levels of parallelism

Other important takeaways

- Gangs can have more than 1 worker and share resources like cache
- Multiple gangs work independently of each other
- Gangs have to be at the outermost level of parallelism
- Vector at the innermost level



OpenACC execution model



- Threads are executed on streaming multiprocessors (SMs)
- Thread blocks do not migrate nor can be split across SMs
- Several concurrent thread blocks can reside on 1 multiprocessor
- A kernel is launched as a grid of thread blocks

Syntax for gang worker vector

- Outermost loop must be a gang
- Innermost loop must be a vector
- A worker loop can appear in between

```
#pragma acc parallel loop gang
for ( i=0; i<N; i++)
  #pragma acc loop vector
  for ( j=0; j<M; j++)
    ;
```

Syntax for gang worker vector

- Additionally, you can specify the no. of gangs, workers and vector length to use for the loop
- Vector of 128 informs the compiler to use a vector length of 128 for the loop

```
#pragma acc kernels
{
#pragma acc loop gang
for ( i=0; i<N; i++)
  #pragma acc loop vector(128)
  for ( j=0; j<M; j++)
    ;
}
```

```
#pragma acc parallel loop gang vector \
             device_type(acc_device_nvidia) vector_length(128) \
             device_type(acc_device_radeon) vector_length(256)
for (i=0; i<N; i++)
{
```

A little detour – vector length advantages

- What are vector processors/vector length?
  - A single vector instruction performs a great deal of work - meaning less fetches and fewer branches (and in turn fewer mispredictions).
  - Vector instructions access memory a block at a time which results in very low memory latency
  - Less memory access = faster processing time.
  - Each result is independent of previous results - allowing high clock rates.

ACK: https://www.cs.uic.edu/~ajayk/c566/VectorProcessors.pdf

A little detour – vector length disadvantages

- What are vector processors NOT good at?
  - Works well only with data that can be executed in highly or completely parallel manner
  - Needs large blocks of data to operate on to be efficient because of the recent advances increasing speed of accessing memory
  - Severely lacking in performance compared to normal processors on scalar data
  - High price of individual chips due to limitations of on-chip memory
  - Increased code complexity needed to vectorize the data
  - High cost in design and low returns compared to superscalar microprocessors

ACK: https://www.cs.uic.edu/~ajayk/c566/VectorProcessors.pdf

OpenACC directives and clauses

Directives

- Parallel
- kernel
- Parallel loop (work sharing directive)

Data Scoping clauses

- copyin
- copyout
- copy
- create
- delete
- present

OpenACC directives and clauses

Directives

- Parallel
- kernel
- Parallel loop (work sharing directive)

Data Scoping clauses

- copyin
- copyout
- copy
- create
- delete
- present

- private
- firstprivate
- num_gangs
- num_workers
- vector_length
- reduction

and more…

Data Construct

- This gives the programmer additional control over how and when data is created and destroyed on a GPU and when data is copied **between CPU and GPU**.
- Without the data directive, OpenACC will make assumptions about whether data is already on the device or not.
- By using the data **construct you help to ensure correctness**, and also improve performance by **avoiding unnecessary data copies**.
- The data directive may be used in conjunction with many other directives **including parallel and loop**.
- The data construct can accept 7 clauses

OpenACC Data Clauses

- **copyin(list)** - Allocates memory on GPU and copies data from host to GPU when entering region.
- **copyout(list)** - Allocates memory on GPU and copies data to the host when exiting region.
- **copy(list)** - Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.
- **create(list)** - Allocates memory on GPU but does not copy.
- **delete(list)** - Deallocate memory on the GPU without copying. (Unstructured Only)
- **present(list)** - Data is already present on GPU from another containing data region.

```
#pragma acc data copyout(a[0:N]), copyin(b[0:N])
{
 #pragma acc parallel loop present(a,b)
 for (int i=0; i<N; i++)
     a[i] = b[i] + 1;
}
```

```
const int N=100;
#pragma acc data copy(a[0:N])
{
 #pragma acc parallel loop present(a)
 for (int i=0; i<N; i++)
     a[i] = a[i] + 1;
}
```

```
#pragma acc data copyout(a[0:N]), create(b[0:N])
{
 #pragma acc parallel loop
 for (int i=0; i<N; i++)
     b[i] = i * 2.0;
```

OpenACC Data Clauses

- **create(list)** - Allocates memory on GPU but does not copy.

```
#pragma acc data copyout(a[0:N]), create(b[0:N])
{
 #pragma acc parallel loop
 for (int i=0; i<N; i++)
     b[i] = i * 2.0;
```

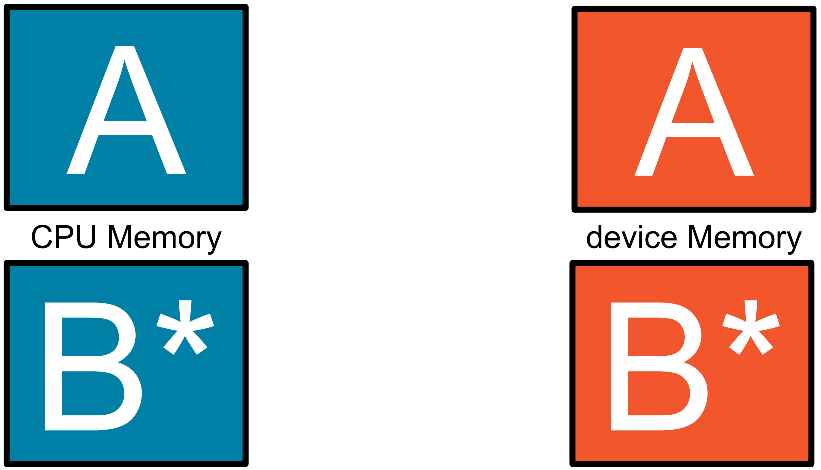
- **present(list)** - Data is already present on GPU from another containing data region.

```
 #pragma acc parallel loop present(a,b)
 for (int i=0; i<N; i++)
     a[i] = b[i] + 1;
}
```

OpenACC update directive

The data must exist on both the CPU and device for the update directive to work.

```
#pragma acc update device(A[0:N])
```




```
#pragma acc update self(A[0:N])
```

OpenACC update directive

update: Explicitly transfers data between the host and the device

Useful when you want to synchronize data in the middle of a data region

Clauses:

- **self:** makes host data agree with device data
- **device:** makes device data agree with host data

```
#pragma acc update self(x[0:count])
#pragma acc update device(x[0:count])
```

REDUCTION CLAUSE OPERATORS

Operator	Description	Example
+	Addition/Summation	reduction(+:sum)
*	Multiplication/Product	reduction(*:product)
max	Maximum value	reduction(max:maximum)
min	Minimum value	reduction(min:minimum)
&	Bitwise and	reduction(&:val)
|	Bitwise or	reduction(|:val)
&&	Logical and	reduction(&&:val)
||	Logical or	reduction(||:val)