# Deep Learning and GPU Programming Workshop

12 – 15  July 2021

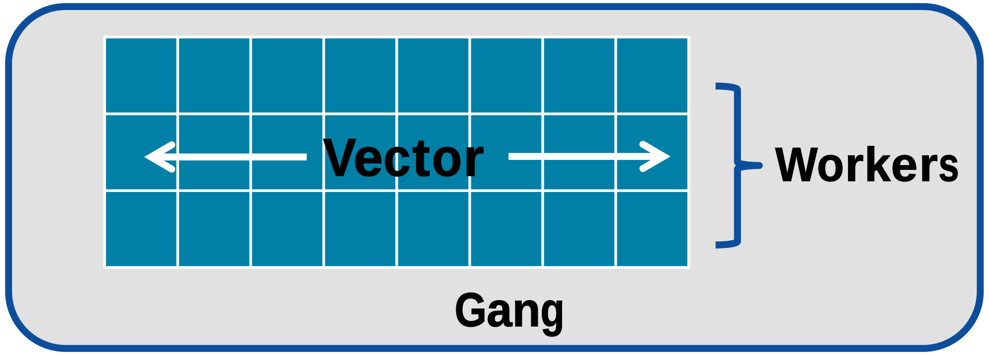
# MODULE SIX:
# LOOP OPTIMIZATIONS

Dr. Volker Weinberg | LRZ | 12.07.2021

# GANG WORKER VECTOR

# GANG WORKER VECTOR

- Gang / Worker / Vector defines the various levels of parallelism we can achieve with OpenACC
- This parallelism is most useful when parallelizing multi-dimensional loop nests
- OpenACC allows us to define a generic Gang / Worker / Vector model that will be applicable to a variety of hardware, but we fill focus a little bit on a GPU specific implementation



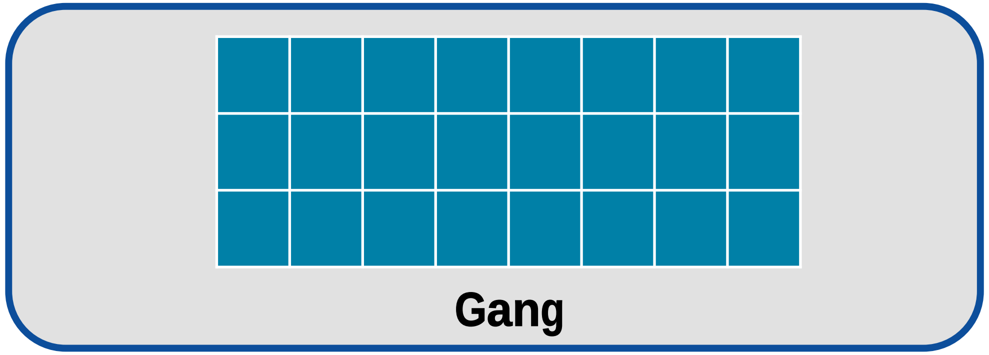
# GANG WORKER VECTOR

- When paralleling our loops, the highest level of parallelism is **gang level parallelism**

- When encountering either the kernels or parallel directive, multiple gangs will be generated, and loop iterations will be spread across the gangs

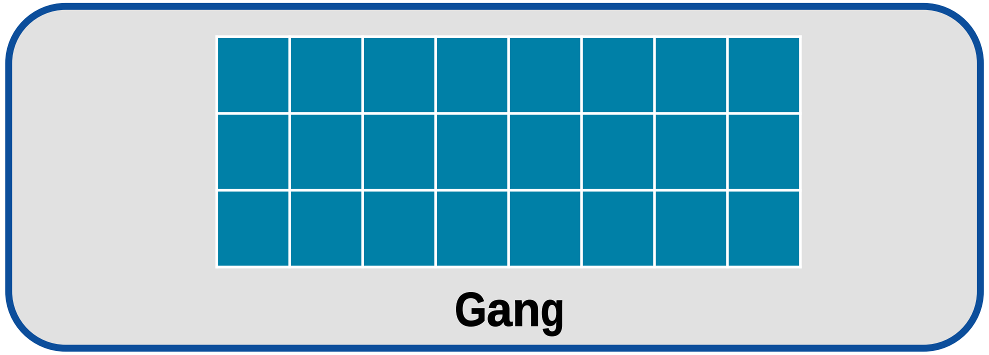
- These gangs are completely independent of each other, and there is no way to for the programmer to know exactly how many gangs are running at a given time

- In many architecures, the gangs have completely separate (or private) memory



# GANG WORKER VECTOR

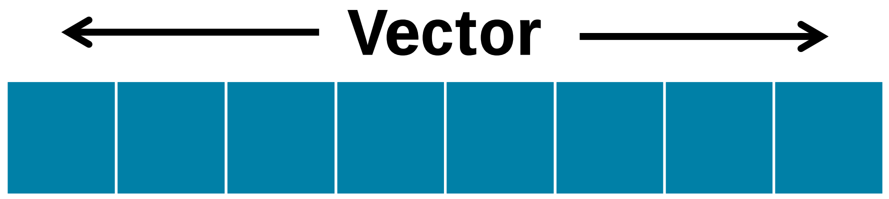
- In our code example, we see that we are applying the **gang** clause to an outer-loop
- This means that the outer-loop iterations will be split across some number of gangs
- These gangs will then execute in parallel with each other
- Whenever a parallel compute region is encountered, some number of gangs will be created
- The programmer is able to specify exactly how many gangs to create



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

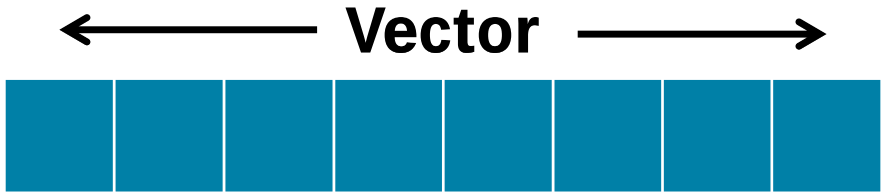
# GANG WORKER **VECTOR**

- A **vector** is the lowest level of parallelism
- Every gang will have **at least 1 vector**
- A vector has the ability to **run a single instruction** on **multiple data elements**
- Many different architectures can implement vectors in different ways, however, OpenACC allows for us to define them in a general, non-hardware-specific way



# GANG WORKER **VECTOR**

- In our code example, the inner-loop iterations will be evenly divided across a vector
- This means that those loop iterations will be executing in parallel with one-another
- Any loop that is **inside** of our vector loop cannot be parallelized further

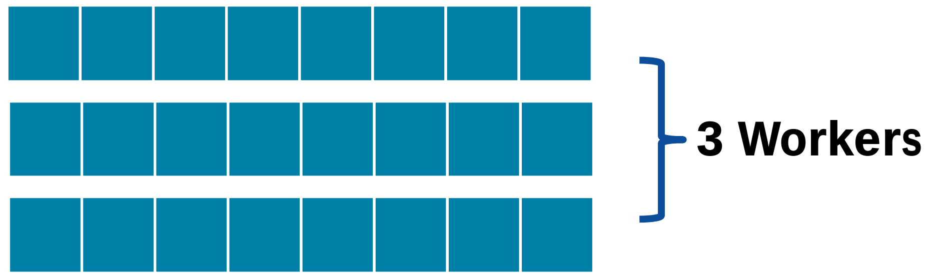


```
#pragma acc parallel loop gang
for( i = 0; i < N; i++ )
  #pragma acc loop vector
  for( j = 0; j < M; j++ )
    < loop code >
```

OpenACC
More Science, Less Programming

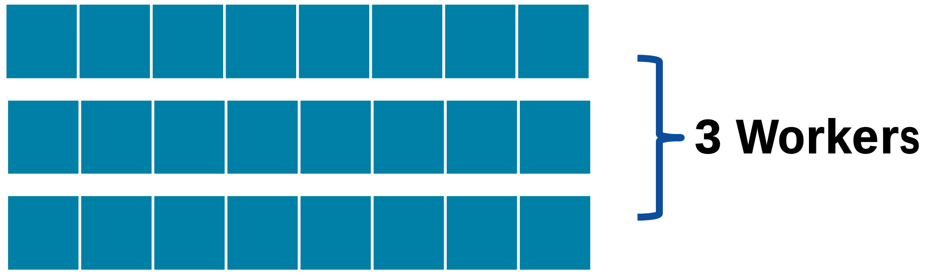
# GANG WORKER VECTOR

- The **worker clause** is a way for the programmer to have **multiple vectors** within a gang
- The primary use of the worker clause is to split up one large vector into multiple smaller vectors
- This can be useful when our inner parallel loops are very small, and will not benefit from having a large vector



# GANG **WORKER** VECTOR

- In our sample code, we apply both gang and worker level parallelism to our outer-loop
- The main difference this creates for our code is that we can now have smaller vectors running the inner loop
- This will most likely improve performance **if** the inner loop is relatively small



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

# PARALLEL DIRECTIVE SYNTAX

- When using the parallel directive, you may define the number of gangs/workers/vectors with **num_gangs(N)**, **num_workers(M)**, **vector_length(Q)**
- Then, you may define where they belong in the loops using **gang**, **worker**, **vector**

```
#pragma acc parallel num_gangs(2) \
 num_workers(2) vector_length(32)
{
  #pragma acc loop gang worker
  for(int x = 0; x < 4; x++){
    #pragma acc loop vector
    for(int y = 0; y < 32; y++){
      array[x][y]++;
    }
  }
}
```

# PARALLEL DIRECTIVE SYNTAX

- You may also apply gang/worker/vector when using the parallel loop construct

```
#pragma acc parallel loop num_gangs(2) num_workers(2) \
  vector_length(32) gang worker
for(int x = 0; x < 4; x++){
  #pragma acc loop vector
  for(int y = 0; y < 32; y++){
    array[x][y]++;
  }
}
```

## KERNELS DIRECTIVE SYNTAX

- When using the kernels directive, the process is somewhat simplified
- You may define the location and number by using **gang(N)**, **worker(M)**, **vector(Q)**
- You may also define gang, worker, and vector using the same method as with the parallel directive
- If you do not specify a number, the compiler will decide one

```
#pragma acc kernels loop gang(2) worker(2)
for(int x = 0; x < 4; x++){
  #pragma acc loop vector(32)
  for(int y = 0; y < 32; y++){
    array[x][y]++;
  }
}
```

OpenACC
More Science, Less Programming

# KERNELS DIRECTIVE SYNTAX

- When using the kernels directive, the process is somewhat simplified
- You may define the location and number by using **gang(N)**, **worker(M)**, **vector(Q)**
- You may also define gang, worker, and vector using the same method as with the parallel directive
- If you do not specify a number, the compiler will decide one
- Each loop nest can have different values for gang, worker, and vector

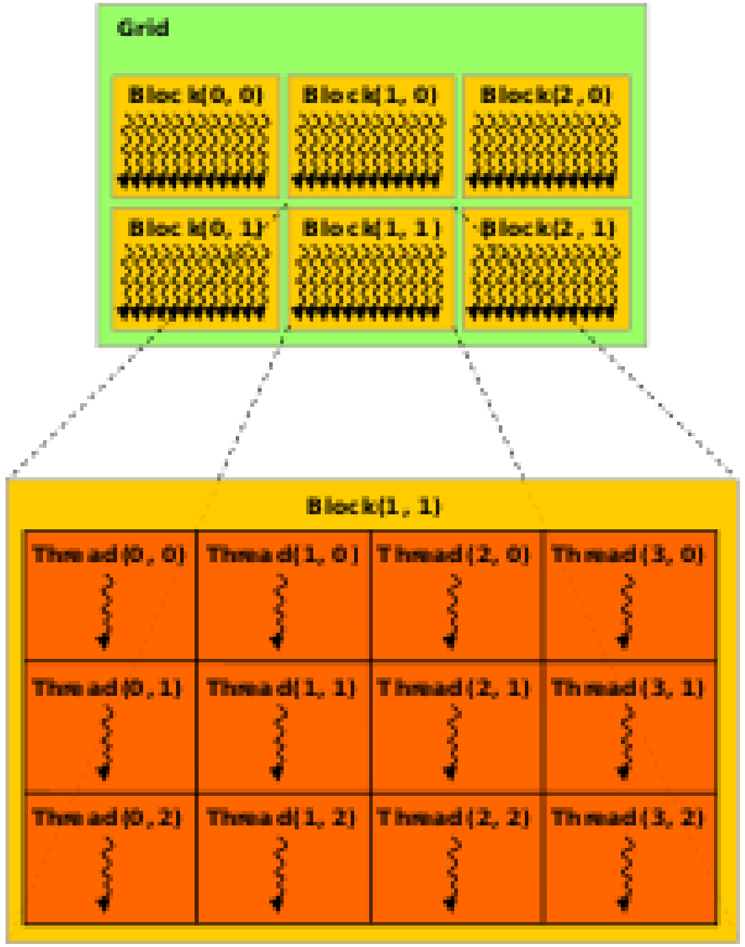
```
#pragma acc kernels
{
  #pragma acc loop gang(2) worker(2)
  for(int x = 0; x < 4; x++){
    #pragma acc loop vector(32)
    for(int y = 0; y < 32; y++){
      array[x][y]++;
    }
  }

  #pragma acc loop gang(4) worker(4)
  for(int x = 0; x < 16; x++){
    #pragma acc loop vector(16)
    for(int y = 0; y < 16; y++){
      array2[x][y]++;
    }
  }
}
```

## WARPS

- So far we have been using a very small number of gangs/worker/vectors, simply because they’re easier to understand
- When actually programming, the number of gangs/worker/vectors will be much larger
- When specifically programming for an NVIDIA GPU, you will always want your vectors large enough to fully utilize **warps**
- A warp, simply put, is an optimized group of 32 threads
- To utilize warps in OpenACC, always make sure that your vector length is a **multiple of 32**

# CUDA PROGRAMMING MODEL REVIEW

- A grid is composed of blocks which are completely independent
- A block is composed of threads which can communicate within their own block
- 32 threads form a warp
- Instructions are issued per warp
- If an operand is not ready the warp will stall
- Context switch between warps when stalled



## GANG WORKER VECTOR

- Gang is a general term that can mean a few different things. In short, it depends on your architecture.
    - On a multicore CPU, generally gang=thread.
    - On a GPU, generally gang=thread block.
- The way I like to think of it is that gang represents my outer-most level of parallelism for any architecture I am running on.

## LOOP OPTIMIZATION RULES OF THUMB

- It is rarely a good idea to set the number of gangs in your code, let the compiler decide.
- Most of the time you can effectively tune a loop nest by adjusting only the vector length.
- It is rare to use a worker loop. When the vector length is very short, a worker loop can increase the parallelism in your gang.
- When possible, the vector loop should step through your arrays
- Use the device_type clause to ensure that tuning for one architecture doesn't negatively affect other architectures.

## MODULE REVIEW

# KEY CONCEPTS

## In this module we discussed…

- The loop directive enables the programmer to give more information to the compiler about specific loops
- This information may be used for correctness or to improve performance.
- The device_type clause allows the programmer to optimize for one device type without hurting others.

## LAB ASSIGNMENT

### In this module's lab you will…

- Update the code from the previous module in attempt to improve the performance
- Use PGProf to analyze the performance difference when changing your loops
- Experiment with the device_type clause to ensure GPU optimizations don't slow down the multicore speed-up, or vice versa