MODULE SIX: LOOP OPTIMIZATIONS

OpenACC

More Science, Less Programming

LOOP OPTIMIZATIONS

- Majority of program runtime is spent in loops
- Every loop can execute in a very different way
- Using OpenACC loop optimization, we can speed-up our most time-consuming portions of code

OpenACC

SAMPLE LOOP CODE

Matrix multiplication

- Our code is a 3-Dimensional Matrix Multiplication code
- The code allows for many different levels and types of parallelism, and works well with all of our loop clauses

```
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    for( k = 0; k < size; k++ )
      c[i][j] += a[i][k] * b[k][j];
```

PARALLELIZING LOOPS

AUTO CLAUSE

- The **auto** clause tells the compiler to decide whether or not the loop is parallelizable
- The auto clause can be very useful when you are unsure of whether or not a loop is safe to parallelize

```
#pragma acc parallel loop auto
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    for( k = 0; k < size; k++ )
      c[i][j] += a[i][k] * b[k][j];
```

AUTO CLAUSE

- When using the **kernels directive**, the auto clause is **implied**
- This means that you do not need to include the auto clause when using the kernels directive
- However, the auto clause can be very useful when using the **parallel directive**

```
#pragma acc kernels loop auto
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    for( k = 0; k < size; k++ )
      c[i][j] += a[i][k] * b[k][j];
```

OpenACC
Directives for Accelerators

INDEPENDENT CLAUSE

- The **independent** clause asserts to the compiler that the loop is parallelizable
- This will overwrite any decision that the compiler makes about the loop
- Adding the independent clause could force the compiler to parallelize a non-parallel loop
- Allows the programmer to force parallelism when using the kernels directive

```
#pragma acc kernels loop independent
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    for( k = 0; k < size; k++ )
      c[i][j] += a[i][k] * b[k][j];
```

INDEPENDENT CLAUSE

- When using the **parallel directive**, the independent clause is **implied**
- With the parallel directive, the programmer is determining which loops are parallelizable and thus the independent clause is not needed

```
#pragma acc parallel loop independent
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    for( k = 0; k < size; k++ )
      c[i][j] += a[i][k] * b[k][j];
```

LOOP CORRECTNESS

SEQ CLAUSE

- The **seq** clause (short for sequential) will tell the compiler to run the loop sequentially
- In the sample code, the compiler will parallelize the outer loops across the parallel threads, but each thread will run the inner-most loop sequentially
- The compiler may automatically apply the seq clause to loops that have too many dimensions

```
#pragma acc parallel loop
for( i = 0; i < size; i++ )
  #pragma acc loop
  for( j = 0; j < size; j++ )
    #pragma acc loop seq
    for( k = 0; k < size; k++ )
      c[i][j] += a[i][k] * b[k][j];
```

PRIVATE AND FIRSTPRIVATE CLAUSES

- The **private** clause allows the programmer to define a list of variables as “thread-private”.
- Each thread will be given a private copy of every variable in the comma-separated list
- **firstprivate** is like private except that the private values are initialized to the same value used on the host. **private** variables are uninitialized.

```
double tmp[3];

#pragma acc kernels loop private(tmp[0:3])
for( i = 0; i < size; i++ )
{
  tmp[0] = <value>;
  tmp[1] = <value>;
  tmp[2] = <value>;
}

// note that the host value of “tmp”
// remains unchanged.
```

PRIVATE AND FIRSTPRIVATE CLAUSES

- Variables in **private** or **firstprivate** clause are private to the loop level on which the clause appears.
- Private variables on an outer loop are shared within inner loops.

```
double tmp[3];

#pragma acc kernels loop private(tmp[0:3])
for( i = 0; i < size; i++ ) {
  // the tmp array is private to each iteration
  // of the outer loop
  tmp[0] = <value>;
  tmp[1] = <value>;
  tmp[2] = <value>;
  #pragma acc loop
  for ( j = 0; j < size2; j++) {
      // but tmp is shared amongst the threads
      // in the inner loop
      array[i][j] = tmp[0]+tmp[1]+tmp[2];
  }
}
```

SCALARS AND PRIVATE CLAUSE

- By default, scalars are **firstprivate** when used in a parallel region and **private** when used in a kernels region.
- Except in some cases, scalars do not need to be added to a private clause. These cases may include but are not limited to:

1. Scalars with global storage such as global variables in C/C++, Module variables in Fortran
2. When the scalar is passed by reference to a device subroutine
3. When the scalar is used as an rvalue after the compute region, aka “live-out”

- Note that putting scalars in a private clause may actually hurt performance!

LOOP OPTIMIZATIONS

COLLAPSE CLAUSE

- **collapse(N)**
- Combine the next N tightly nested loops
- Can turn a multidimensional loop nest into a single-dimension loop
- This can be extremely useful for increasing memory locality, as well as creating larger loops to expose more parallelism

```
#pragma acc parallel loop collapse(2)
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    double tmp = 0.0f;
    #pragma acc loop reduction(+:tmp)
    for( k = 0; k < size; k++ )
      tmp += a[i][k] * b[k][j];
    c[i][j] = tmp;
```

COLLAPSE CLAUSE

collapse(2)

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

```
#pragma acc parallel loop collapse(2)
for( i = 0; i < 4; i++ )
  for( j = 0; j < 4; j++ )
    array[i][j] = 0.0f;
```

TILE CLAUSE

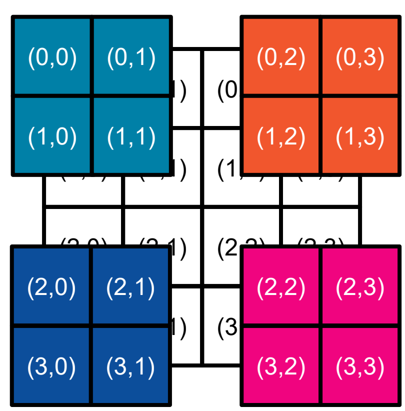
- **tile (x , y , z, ...)**
- Breaks multidimensional loops into “tiles” or “blocks”
- Can increase data locality in some codes
- Will be able to execute multiple “tiles” simultaneously

```
#pragma acc kernels loop tile(32, 32)
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    for( k = 0; k < size; k++ )
      c[i][j] += a[i][k] * b[k][j];
```

TILE CLAUSE

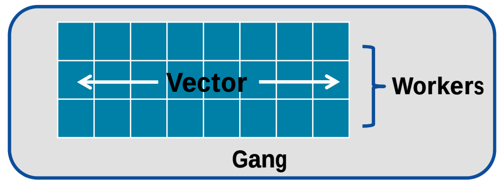
```
#pragma acc kernels loop tile(2,2)
for(int x = 0; x < 4; x++){
  for(int y = 0; y < 4; y++){
    array[x][y]++;
  }
}
```

tile (2 , 2)



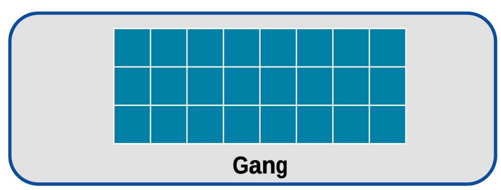
GANG WORKER VECTOR

- 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



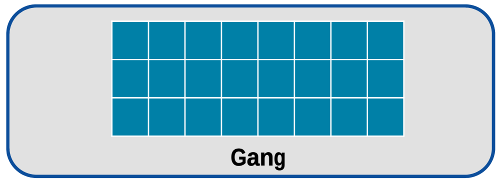
GANG WORKER VECTOR

- 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

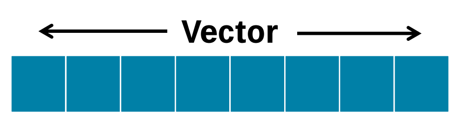
- In our code example, we see that we are applying the **gang** clause to an outer-loop
- This means that the outer-loop iterations will be split across some number of gangs
- These gangs will then execute in parallel with each other
- Whenever a parallel compute region is encountered, some number of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

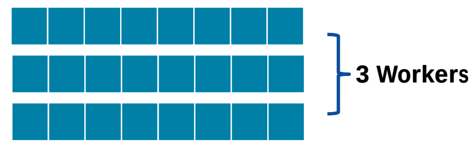

```
#pragma acc parallel loop gang
for( i = 0; i < N; i++ )
  #pragma acc loop vector
  for( j = 0; j < M; j++ )
    < loop code >
```

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]++;
  }
}
```

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

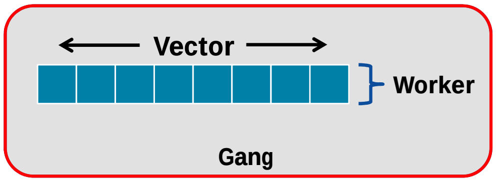
```
#pragma acc kernels
{
  #pragma acc loop gang(2) worker(2)
  for(int x = 0; x < 4; x++){
    #pragma acc loop vector(32)
    for(int y = 0; y < 32; y++){
      array[x][y]++;
    }
  }

  #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]++;
    }
  }
}
```

GANG WORKER VECTOR

```
#pragma acc kernels loop gang worker(1)
for(int x = 0; x < 4; x++){
  #pragma acc loop vector(8)
  for(int y = 0; y < 8; y++){
    array[x][y]++;
  }
}
```

- We have a simple 2-dimensional loop nest
- We have specified that there is **1 worker** and a **vector length of 8**
- We do not specify how many **gangs** to generate, so the compiler will create **enough gangs to cover the loop**

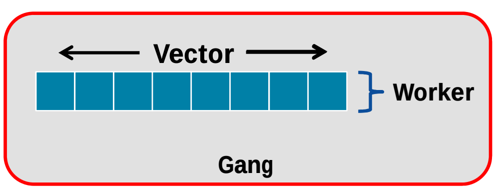
GANG WORKER VECTOR



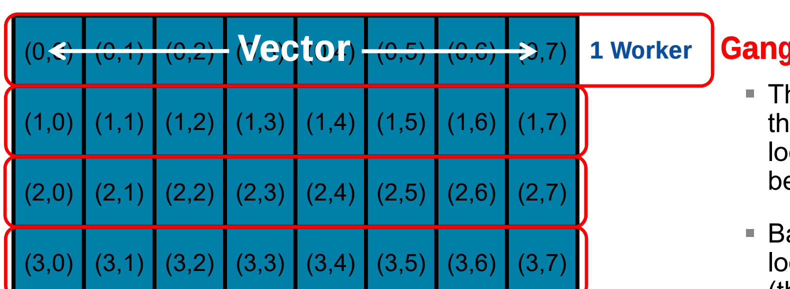

```
#pragma acc kernels loop gang worker(1)
for(int x = 0; x < 4; x++){
  #pragma acc loop vector(8)
  for(int y = 0; y < 8; y++){
    array[x][y]++;
  }
}
```

- The diagram shows a single gang, though the compiler will be able to generate as many gangs as it wants
- These gangs are completely separate from each other, and are indistinguishable
- We will show these gangs apply to a physical loop diagram, but this representation may not be 100% accurate to what the compiler might decide

GANG WORKER VECTOR

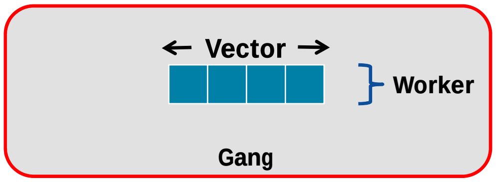



```
#pragma acc kernels loop gang worker(1)
for(int x = 0; x < 4; x++){
  #pragma acc loop vector(8)
  for(int y = 0; y < 8; y++){
    array[x][y]++;
  }
}
```



- The vectors are colored, so that we can observe which loop iterations they are being applied to
- Based on the size of this loop nest, the compiler will (theoretically) generate **4 gangs**

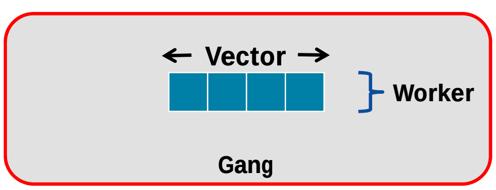
GANG WORKER VECTOR



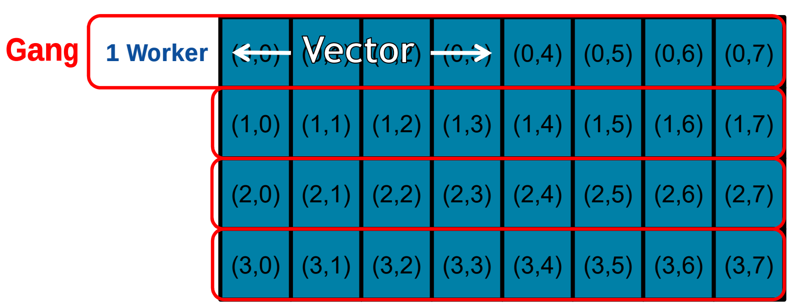

```
#pragma acc kernels loop gang worker(1)
for(int x = 0; x < 4; x++){
  #pragma acc loop vector(4)
  for(int y = 0; y < 8; y++){
    array[x][y]++;
  }
}
```

- We have now reduced the **vector length to 4,** but have kept everything else the same
- The dimension of the outer-loop is still the same, and is still being distributed across the gangs, so the numbers of gangs will not change
- Let's observe how our code will function with a **smaller vector size**

GANG WORKER VECTOR

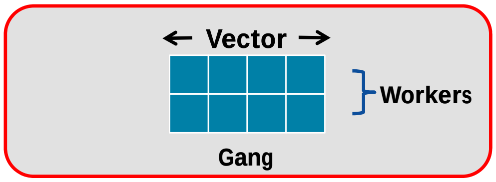



```
#pragma acc kernels loop gang worker(1)
for(int x = 0; x < 4; x++){
  #pragma acc loop vector(4)
  for(int y = 0; y < 8; y++){
    array[x][y]++;
  }
}
```



- We are still generating 4 gangs, but now each vector is computing two loop iterations
- If we wanted to generate **more gangs**, we would need to increase the size of the outer-loop

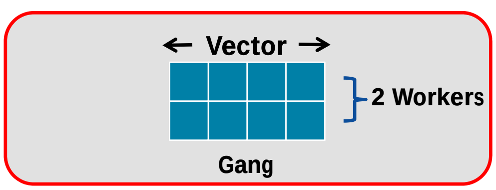
GANG WORKER VECTOR



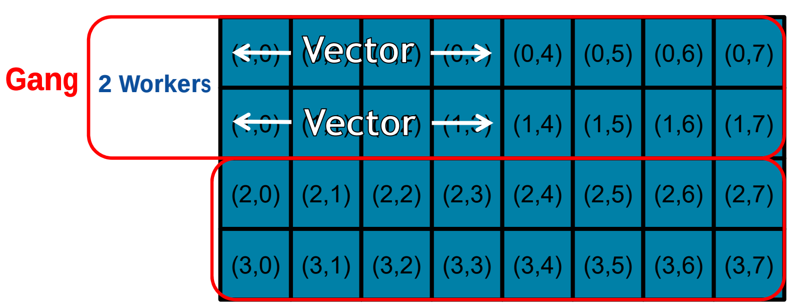

```
#pragma acc kernels loop gang worker(2)
for(int x = 0; x < 4; x++){
  #pragma acc loop vector(4)
  for(int y = 0; y < 8; y++){
    array[x][y]++;
  }
}
```

- For our last trivial example, let's increase the **number of workers to 2**
- There are now **two vectors per gang**, and each **vector is of length 4**

GANG WORKER VECTOR

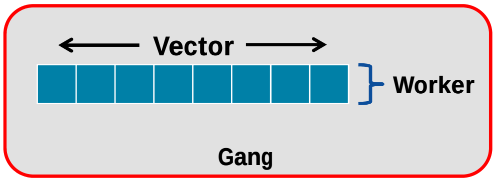



```
#pragma acc kernels loop gang worker(2)
for(int x = 0; x < 4; x++){
  #pragma acc loop vector(4)
  for(int y = 0; y < 8; y++){
    array[x][y]++;
  }
}
```



- Since we have increased the number of workers, we will now only generate **2 gangs**

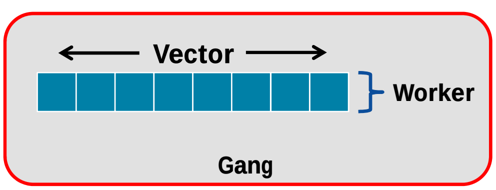
GANG WORKER VECTOR



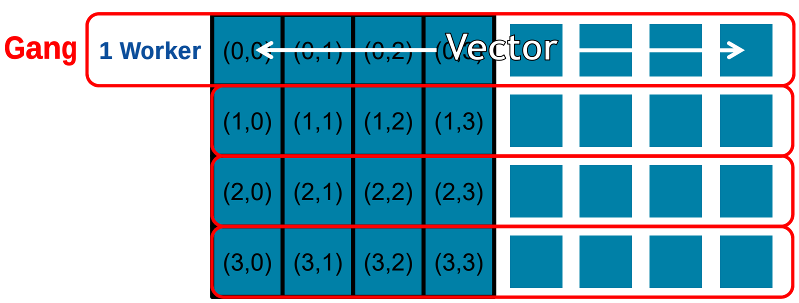

```
#pragma acc kernels loop gang worker(1)
for(int x = 0; x < 4; x++){
  #pragma acc loop vector(8)
  for(int y = 0; y < 4; y++){
    array[x][y]++;
  }
}
```

- Now let's look at a situation where the gang/worker/vector model is very useful
- We have reduced the size of our inner-loop to 4 iterations
- Let's try to run this loop with a vector length of 8

GANG WORKER VECTOR

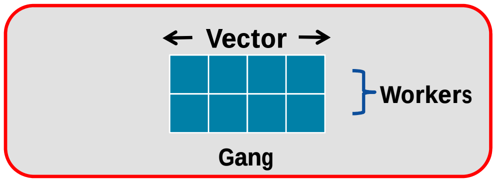



```
#pragma acc kernels loop gang worker(1)
for(int x = 0; x < 4; x++){
  #pragma acc loop vector(8)
  for(int y = 0; y < 4; y++){
    array[x][y]++;
  }
}
```



- We can see that our vector length is **much larger** than our inner-loop
- We are **wasting** half of our vector, meaning our code is performing half as well as it could

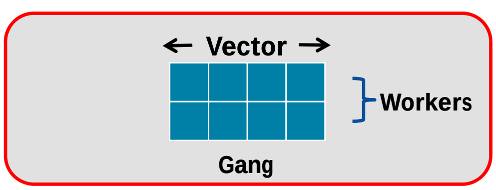
GANG WORKER VECTOR



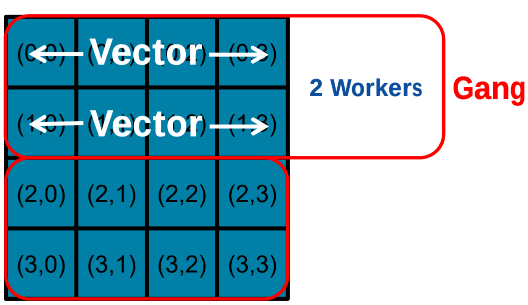

```
#pragma acc kernels loop gang worker(2)
for(int x = 0; x < 4; x++){
  #pragma acc loop vector(4)
  for(int y = 0; y < 4; y++){
    array[x][y]++;
  }
}
```

- We can fix this by **breaking our vector** up among **2 workers**
- Now instead of having 1 long vector, we have 2 shorter vectors
- This setup should fit the organization of our loop better

GANG WORKER VECTOR

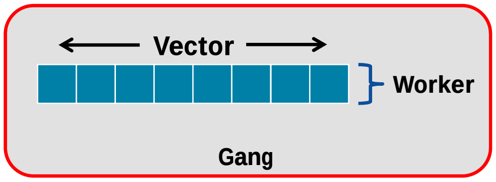



```
#pragma acc kernels loop gang worker(2)
for(int x = 0; x < 4; x++){
  #pragma acc loop vector(4)
  for(int y = 0; y < 4; y++){
    array[x][y]++;
  }
}
```



- We are no longer wasting a portion of our vectors, since the smaller vector size now fits our loop properly
- We always need to consider the size of the loop when choosing the gang worker vector dimensions

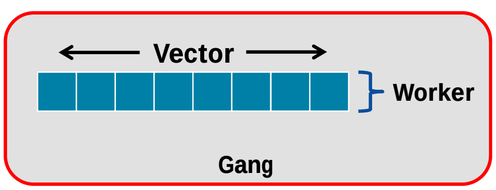
GANG WORKER VECTOR



- Another way we could have fixed this problem is by using the **collapse clause**

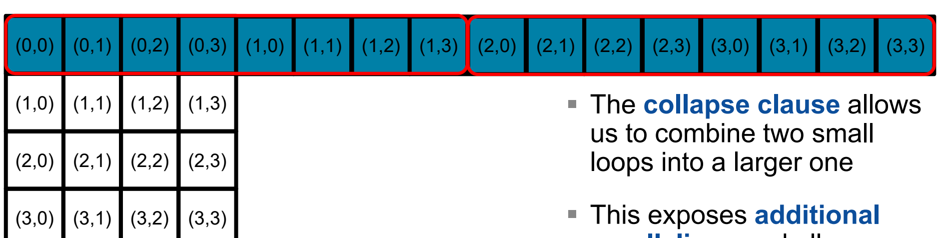
```
#pragma acc kernels loop collapse(2) gang worker(1) vector(8)
for(int x = 0; x < 4; x++){
  for(int y = 0; y < 4; y++){
    array[x][y]++;
  }
}
```

GANG WORKER VECTOR




```
#pragma acc kernels loop collapse(2) \
  gang worker(1) vector(8)
for(int x = 0; x < 4; x++){
  for(int y = 0; y < 4; y++){
    array[x][y]++;
  }
}
```

`collapse( 2 )`



- The **collapse clause** allows us to combine two small loops into a larger one
- This exposes **additional parallelism,** and allows us to use a **longer vector**

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**

DEVICE_TYPE CLAUSE

- **`device_type ( <type> )`**
- Clauses that follow only apply to the specified device type.
- This allows you to optimize for one type (GPU) without hurting the performance of another (CPU)
- Multiple device types can be specified on a single directive.

```
#pragma acc parallel loop collapse(3)\
  device_type(nvidia) \
  vector_length(256)
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    for( k = 0; k < size; k++ )
      c[i][j] += a[i][k] * b[k][j];
```

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