Fundamentals of Accelerated computing

OpenACC

Nitin Shukla

HPC Application Engineer

December 17-18th 2024

NVIDIA DEEP LEARNING INSTITUTE

UNIVERSITÀ DI TRENTO

CINECA

Contents: topics explored

1

What and where will you run an application?

OpenACC labs on NVIDIA platform, Six modules

2

OpenACC directives

Profiling and Parallelising with OpenACC Toolkit on NVIDIA GPUs

3

Data Management

Explicit data management, Data regions and clause, Unstructured Data Lifetimes

4

Loop Optimisation

Overlapping kernel execution & data transfer on Single/Multi GPU

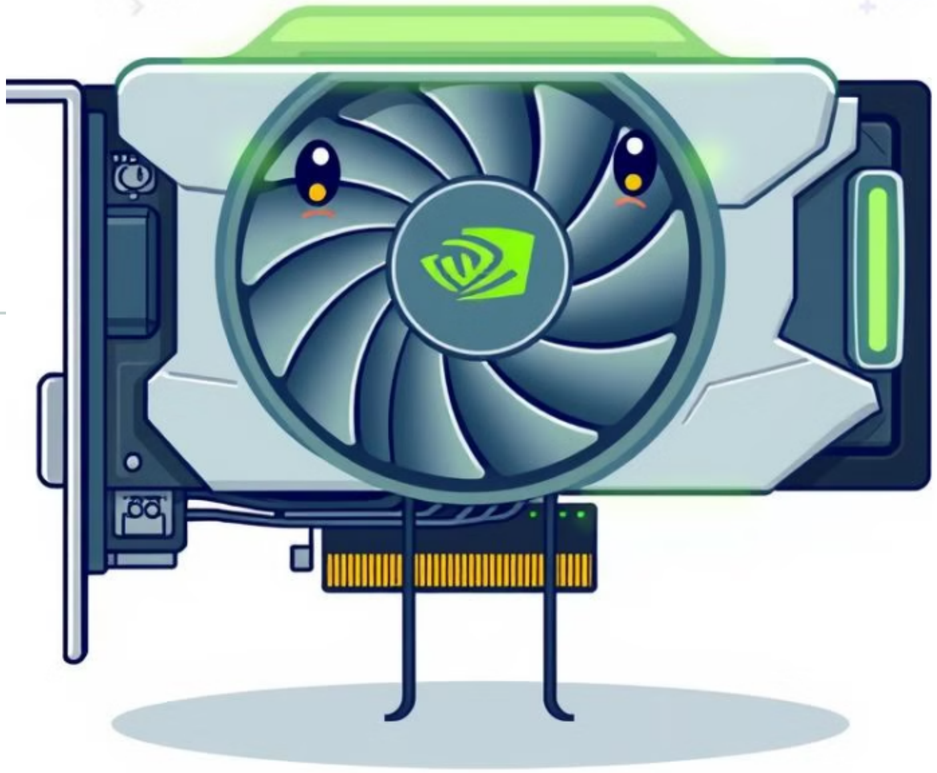
What and where will you run an application ?

Access to NVIDIA platform

Event code:
UTRENTO_OPENACC_AMBASSADOR_DE24

1

Case study: Parallelise a serial Laplace 2D

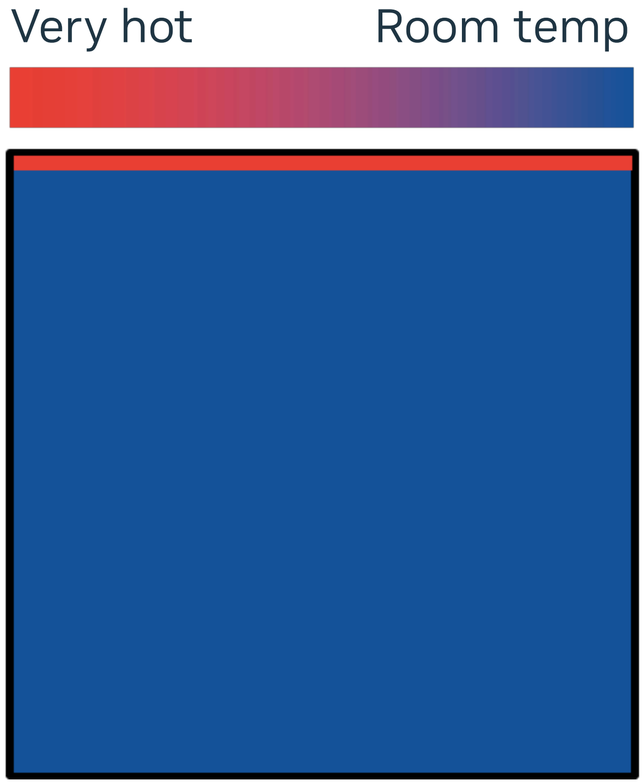


Laplace heat transfer

We will observe a simple simulation of heat distributing across a metal plate.

We will apply a consistent heat to the top of the plate.

Then, we will simulate the heat distributing across the plate.

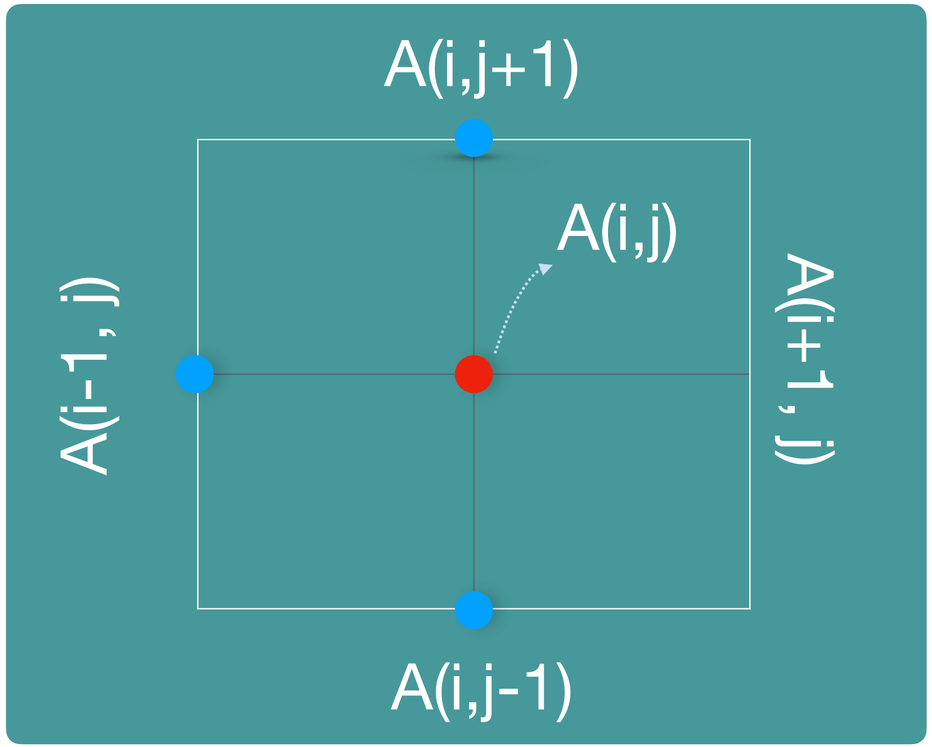


Code description

- Iteratively converges to correct value (e.g. Temperature)
- by computing new values at each point from the average of neighboring points.
- Example: Solve Laplace equation in 2D

$$\nabla^2 f(x,y) = 0$$

$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$



Code description

```
while (error > tol && niter < niter_max)
error = 0.0;

 for (int j = 1; j < n-1; ++j) {
  for (int i = 1; i < m-1; ++i) {
   Anew[idx] = 0.25 * ( A[idx+1] + A[idx-1] + A[idx-m]
   + A[idx+m]);

   error = fmax(error, fabs(Anew[idx] - A[idx])); }
   }
 for (int j = 1; j < n-1; ++j)
  for (int i = 1; i < m-1; ++i)
   A[j][i] = Anew[j][i]
```

Iterate until converged

Iterate cross matrix elements

Compute max error for converge

Compute max error for converge

Swap input/output arrays

NVIDIA's HPC compilers (AKA PGI)

NVIDIA compiler Names (PGI may still work)

nvc - The command to compiler C code (pgcc)

nvc++ - The command to compiler C++ code (pgc++)

nvfortran - The command to compiler Fortran code (pgfortran)

The -fast flag instructs the compiler to optimise the code to the best of its abilities

- nvc -fast my_program.c
- nvc++ -fast my_program.cpp
- nvfortran -fast my_program.cf90

NVIDIA’s HPC compilers (AKA PGI)

Instruct compiler to print feedback about the compiled code

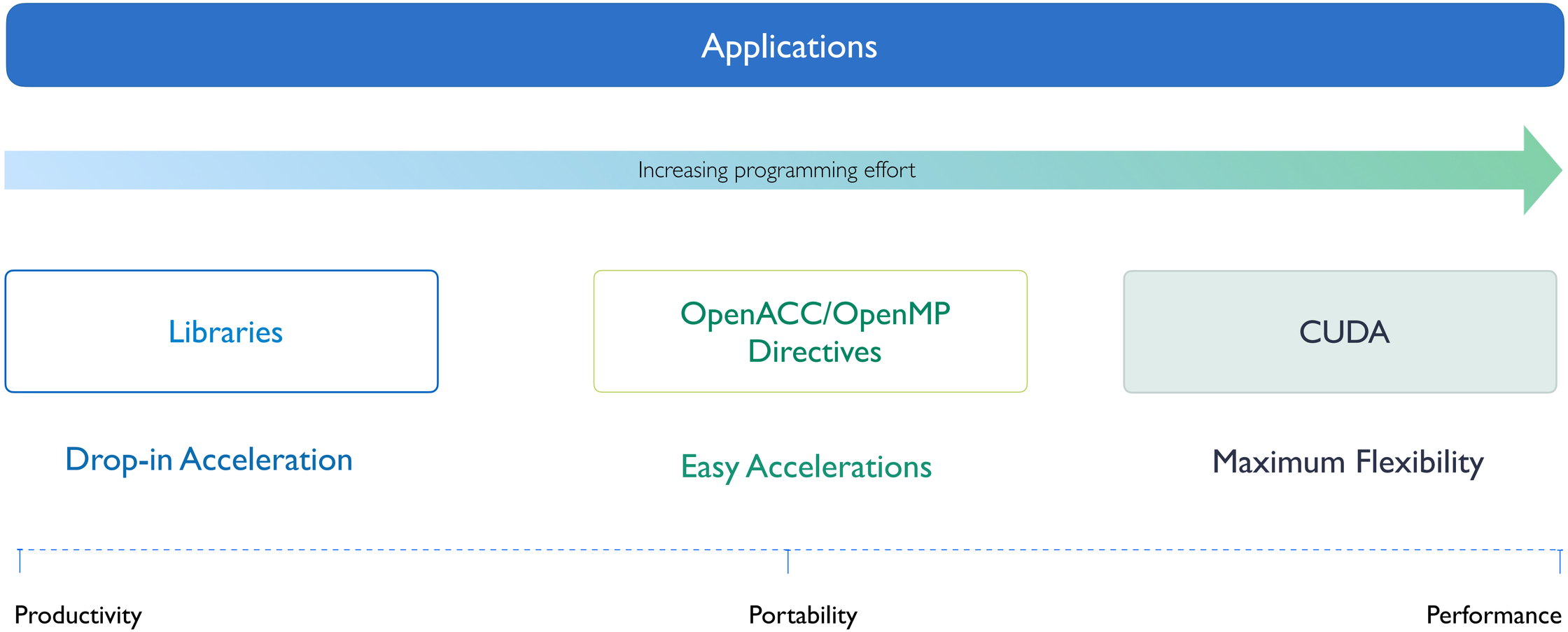
-Minfo = accel informs about what parts of the code were accelerated via OpenX

-Minfo = opt informs about all code optimisations

-Minfo=all gives all code feedback, whether positive or negative

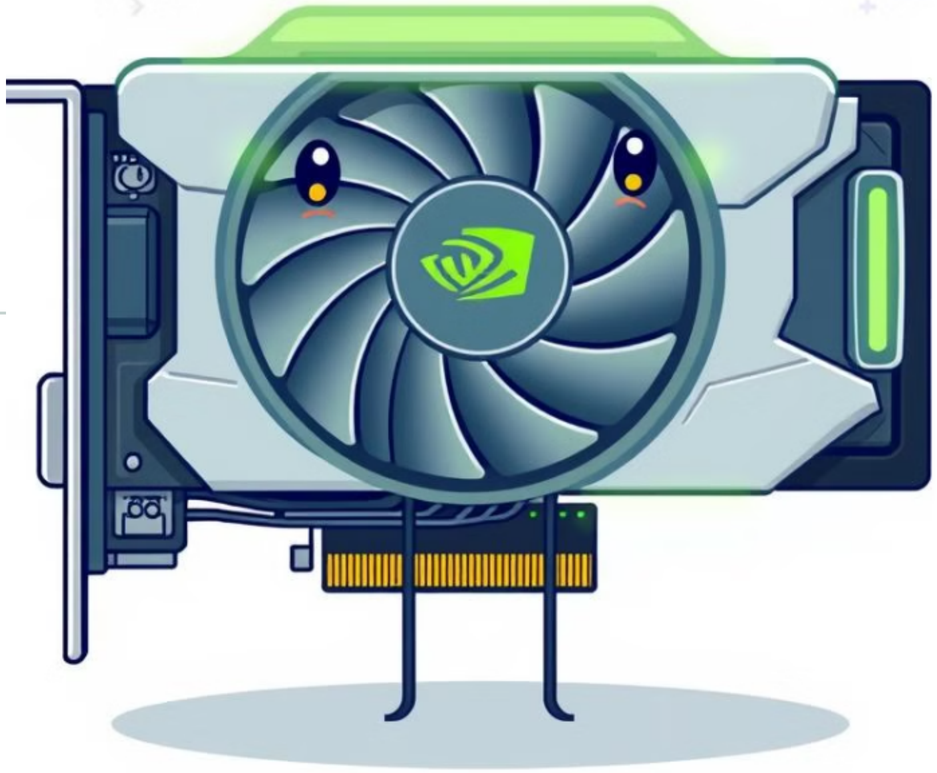
- nvc -fast -Minfo=all my_program.c
- nvc++ -fast -Minfo=all my_program.cpp
- nvfortran -fast -Minfo=all my_program.cf90

Ways to parallels an applications on Nvidia GPUs



1

Why and what are the GPUs?



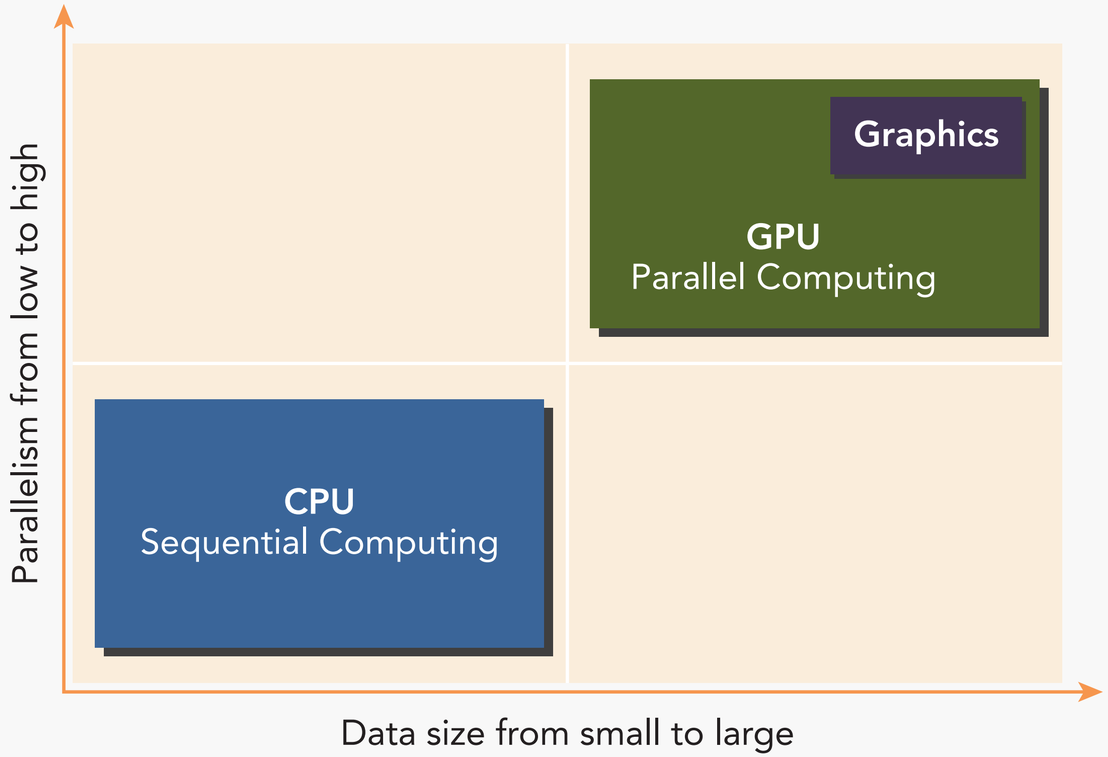
Computer architecture drives parallelism at the core level

Fundamentals types of parallelism

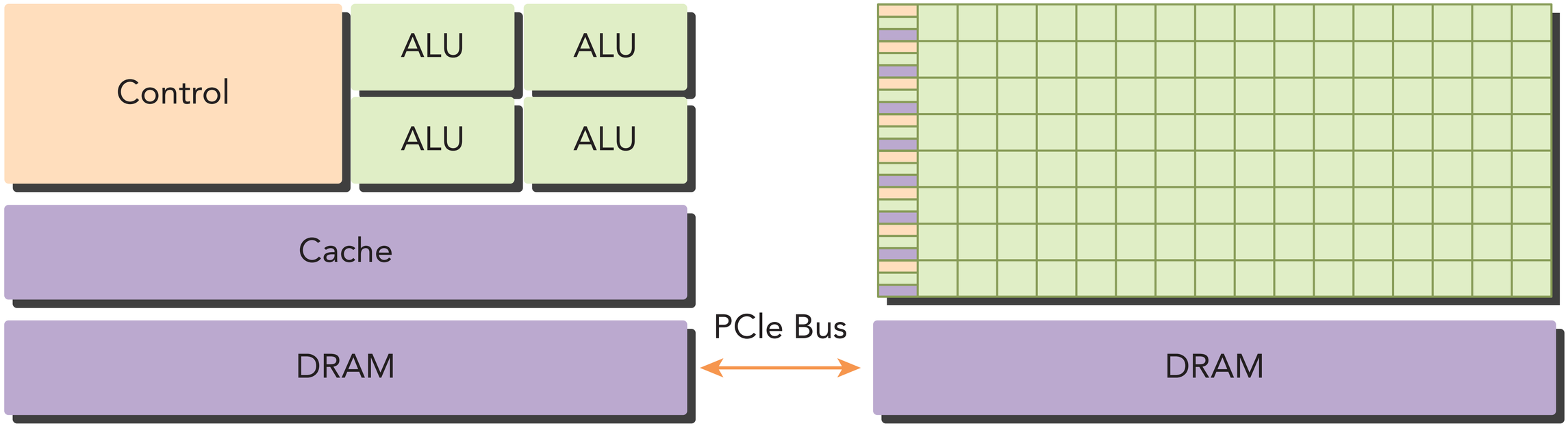
- Task parallelisms: multiple independent tasks can run simultaneously, distributing functions across multiple cores
- Data parallelisms: multiple data items can be processed simultaneously, distributing the data across multiple cores

Heterogeneous computing

- CUDA programming: well-suited to address problems that can be expressed as data-parallel computations



How GPUs are different than CPUs?



CPU (host): minimize latency

GPU(Device): maximize throughput

GPU acceleration for data-parallel tasks

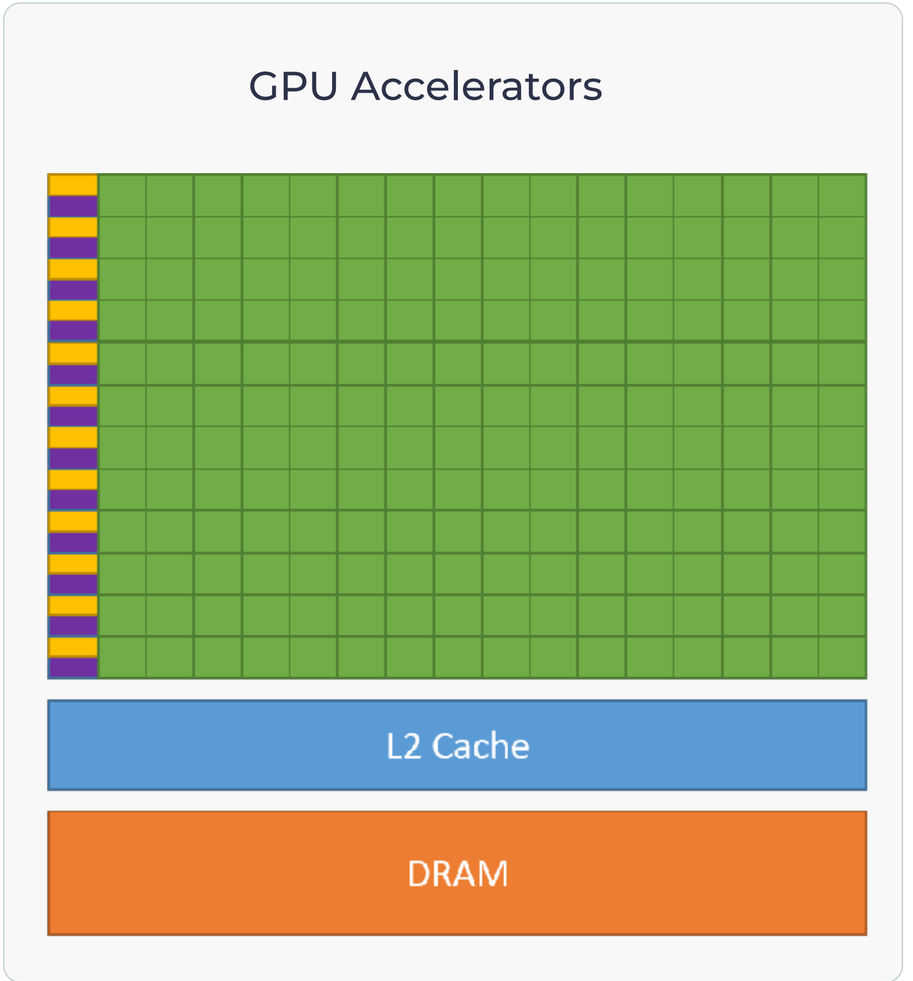
Two important features that describe GPU capability

- Number of CUDA cores
- Memory size

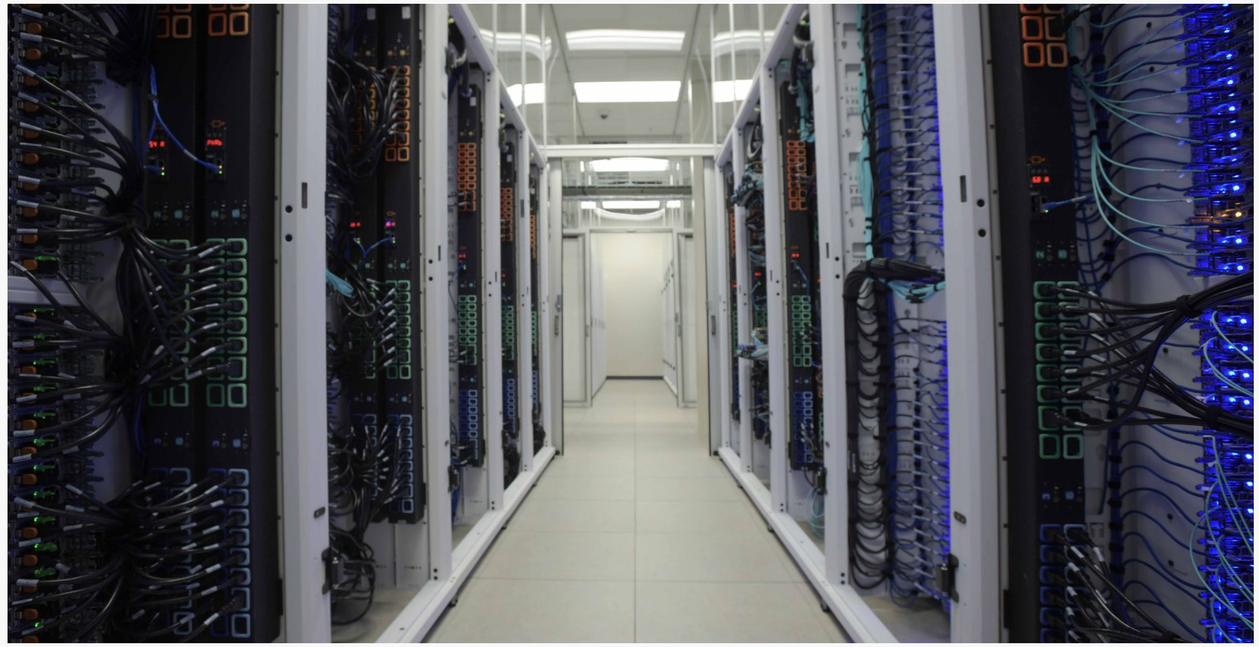
GPU Performance Metrics: Throughput vs. Latency

- Peak computational performance
  measures in Tflops or Pflops, reflects a device's ability to perform floating-point calculations rapidly and efficiently

- Memory bandwidth
  the rate at which data can be transferred between the CPU and memory, measured in gigabytes per second (GB/s). It directly impacts the speed of data-intensive applications.



NVIDIA Tesla A100 with 54 Billion Transistor



- With 7nm technologies
- 19.5 teraflops of FP32 performance
- 6912 CUDA cores, 40GB of graphics memory, and 1.6TB/s of graphics memory bandwidth
- The A100 80GB model announced in Nov 2020, has 2.0TB/s graphics memory bandwidth

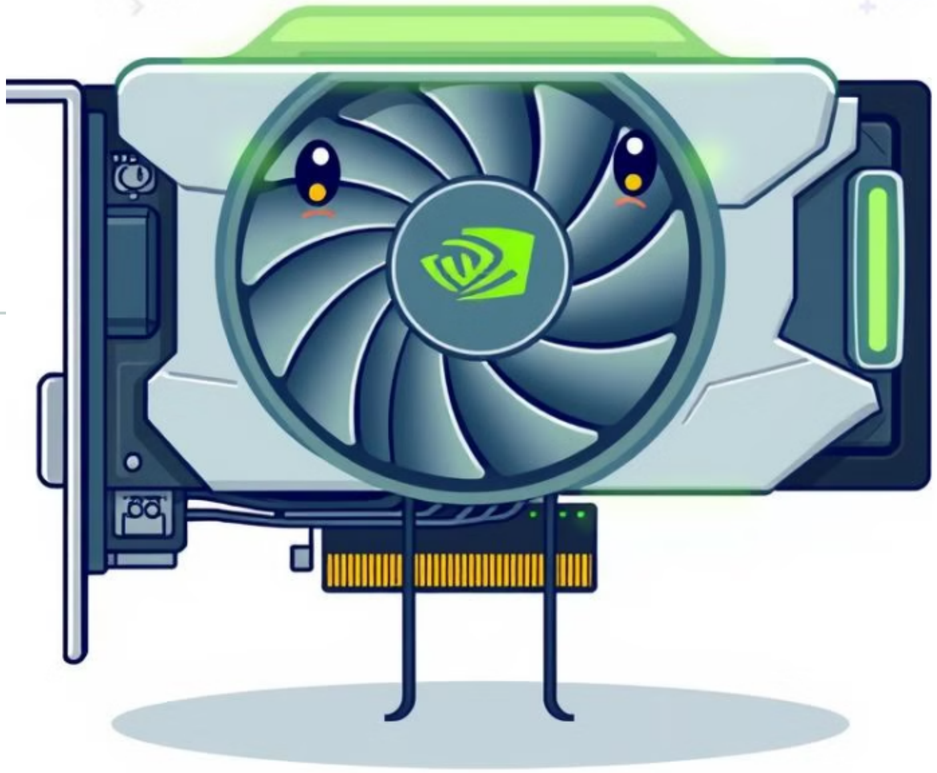
Course objectives:

Enable you to accelerate your application with OpenACC

Course objectives:

2

Enable you to accelerate your application with OpenACC



Development cycle

Analyze

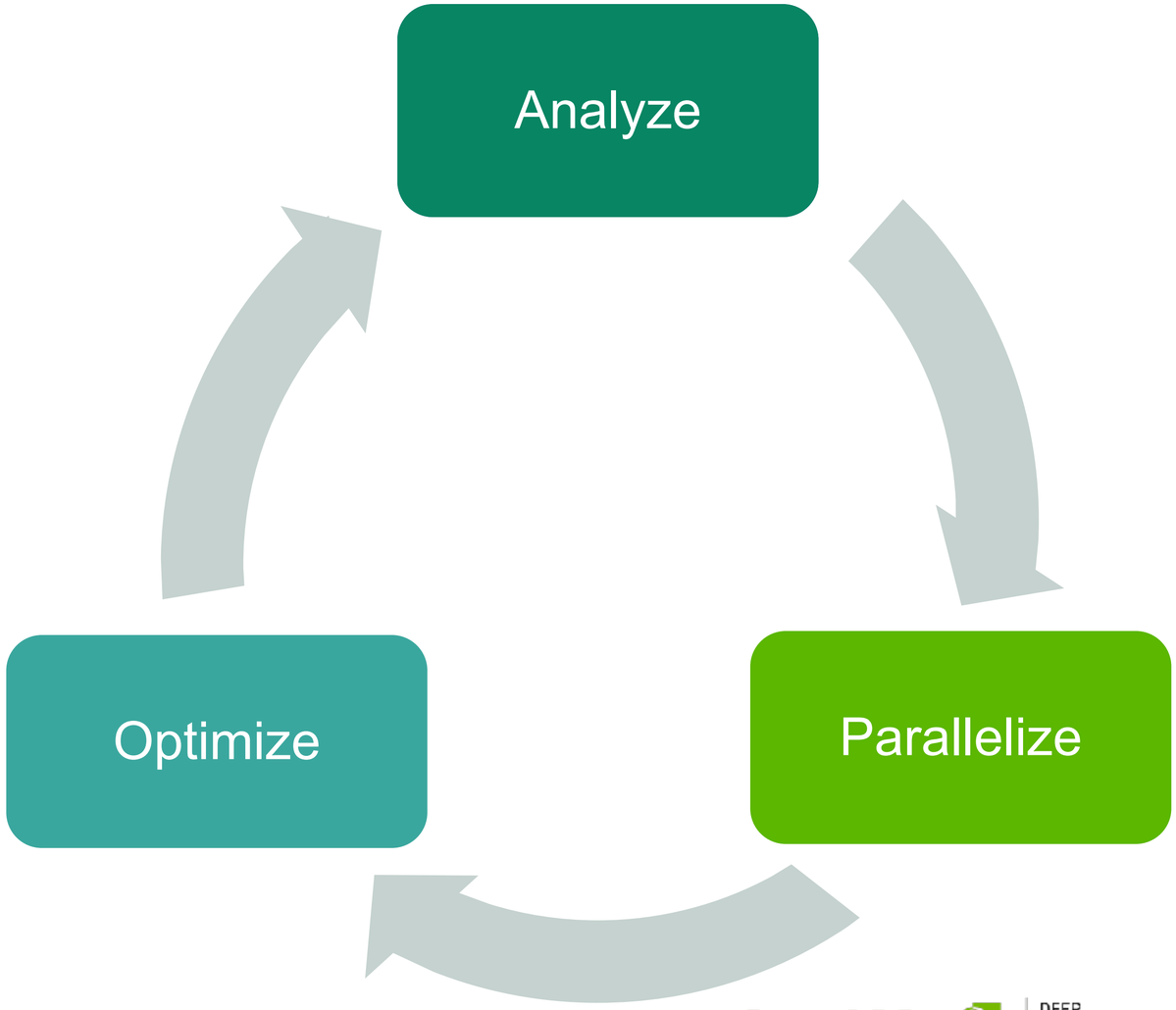
- Your code to determine most likely places needing parallelisation or optimisation

Parallelise

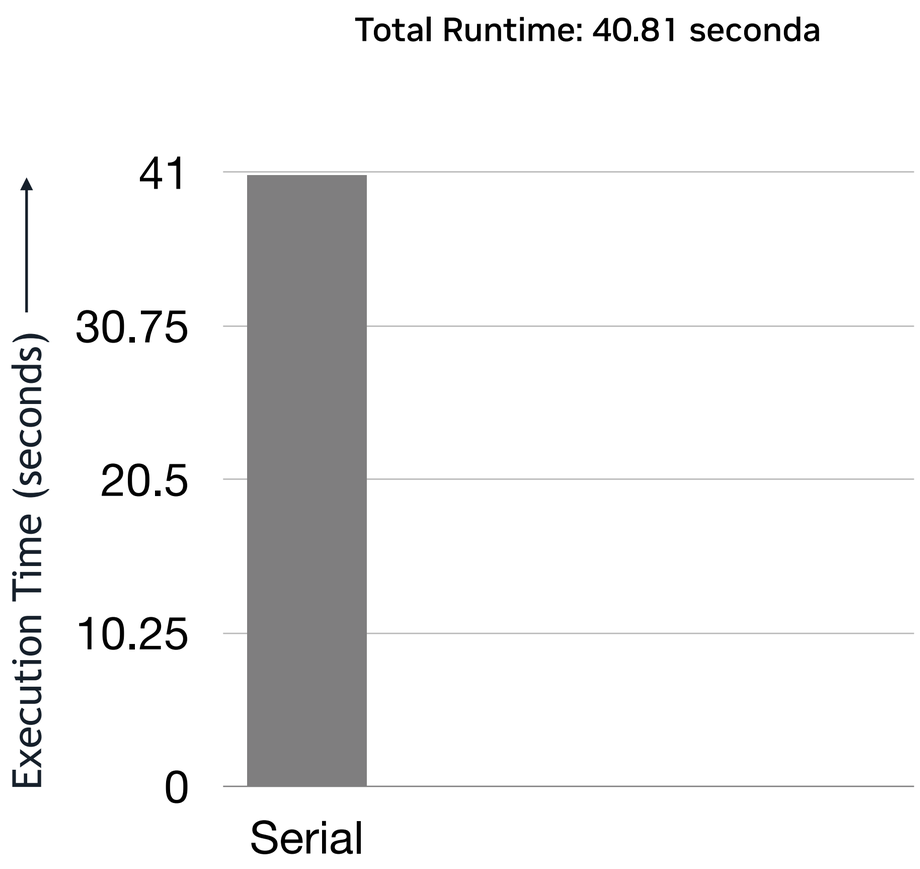
- Your code by starting with the most time consuming parts, check the correctness and analysis again

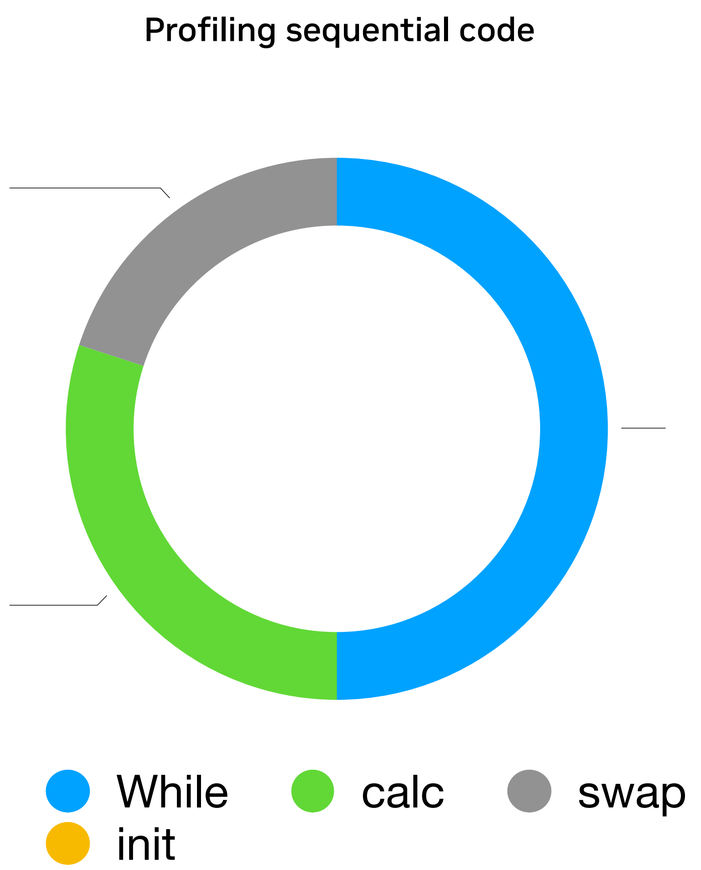
Optimize

- Your code to improve and observed speedup from parallelisation



Hotspots: Identify the portions of code that took the longest to run





OpenMP implementation

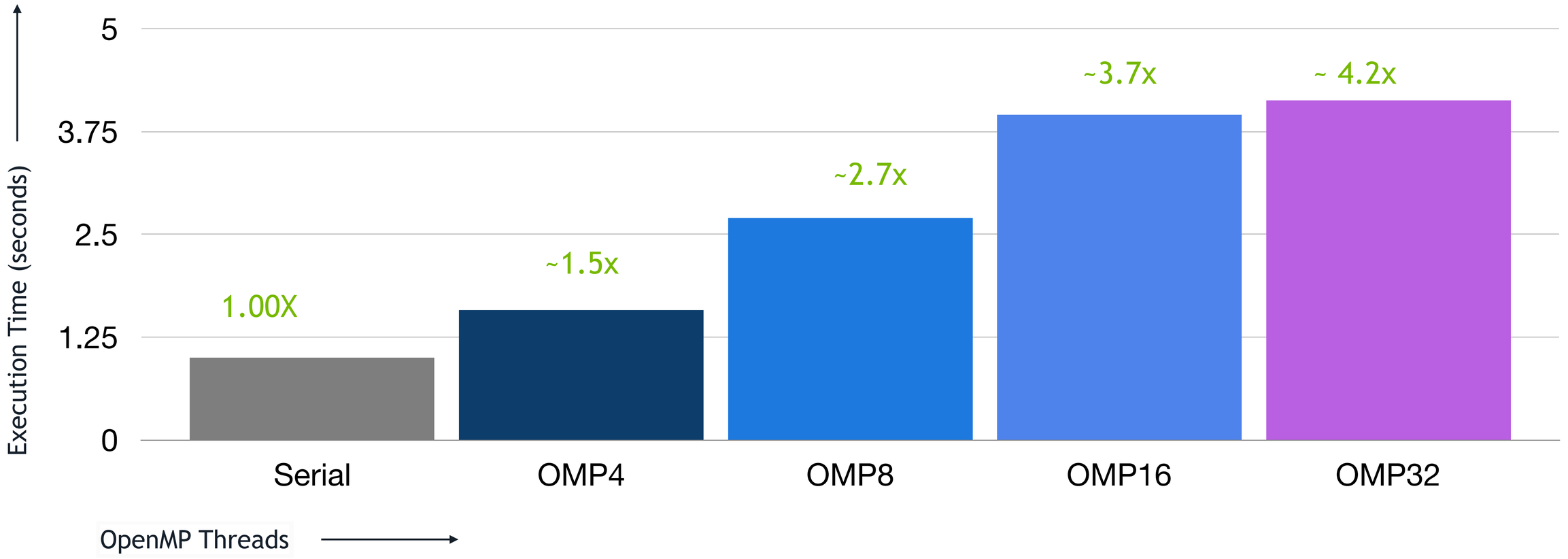
```
while ( error > tol && iter < iter_max ) {
  error=0.0;
#pragma omp parallel for shared(m, n, Anew, A) reduction(max:error)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }
#pragma omp parallel for shared(m, n, Anew, A)
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

OpenMP implementation

```
while ( error > tol && iter < iter_max ) {
  error=0.0;
#pragma omp parallel for collapse(2) shared(m, n, Anew, A) reduction(max:error)
  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]);
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }
#pragma omp parallel for collapse(2) shared(m, n, Anew, A)
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

Performance speed up (higher is better)

Simulation was performed 1000 Iterations



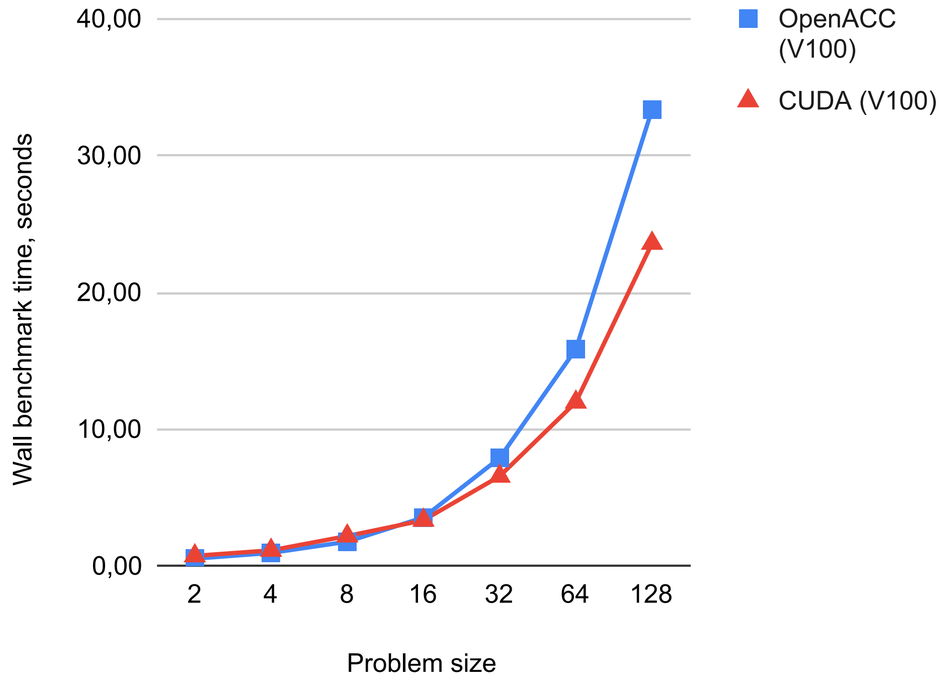
Why OpenACC directives?

OpenACC Directives: Simple, Powerful and Portable

```
main()
 {
   <serial code>
   #pragma acc kernels // Automatically runs on GPU
   <parallel code>
 }
```

OpenACC VS CUDA

- Powerful: Could reach to 98 % speed in comparison to CUDA
- A rough estimate 10,0000 + Developers using OpenACC
- Scientific research, Industry adoption
- Tool and Ecosystem Support: NVHPC, CRAY, LLVM



Comparison of OpenACC and CUDA performance on Cloverleaf application (doi:10.1088/1742-6596/1740/1/012056)

OpenACC Directives

Simple | Powerful | Portable

- Incremental
- Single Source
- Interoperable
- Performance portable: CPU, GPU, MIC

1. Management Data Movement
2. Initiate Parallel Execution
3. Optimize Loop Mappings

```
#pragma acc data copyin(a,b) copycout(c)
{

  ...
  #pragma acc parallel
  {

  #pragma acc loop gang vector
   for (int idx=0; idx<N; idx++)
       C[idx] = A[idx] + B[idx];
  }

}
```

What is OpenACC directives?

OpenACC Directives: Simple, Powerful and Portable

OpenACC is …

a directive-based

parallel programming model

designed for

performance and portability

Add Simple Compiler Directive

```
main()
 {
  <serial code>
  #pragma acc kernels
  {
    <parallel code>
  }
}
```

OpenACC Syntax

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

#pragma in C/C++ is what's known as a "compiler hint." These are very similar to programmer comments, however, the compiler will actually read our pragmas. Pragmas are a way for the programmer to "guide" the compiler, without running the chance damaging the code. If the compiler does not understand the pragma, it can ignore it, rather than throw a syntax error.

acc is an addition to our pragma. It specifies that this is an **OpenACC pragma**. Any non-OpenACC compiler will ignore this pragma. Even the nvc/nvc++ compiler can be told to ignore them. (which lets us run our parallel code sequentially!)

directives are commands in OpenACC that will tell the compiler to do some action. For now, we will only use directives that allow the compiler to parallelize our code.

clauses are additions/alterations to our directives. These include (but are not limited to) optimizations. The way that I prefer to think about it: directives describe a general action for our compiler to do (such as, paralellize our code), and clauses allow the programmer to be more specific (such as, how we specifically want the code to be parallelized).

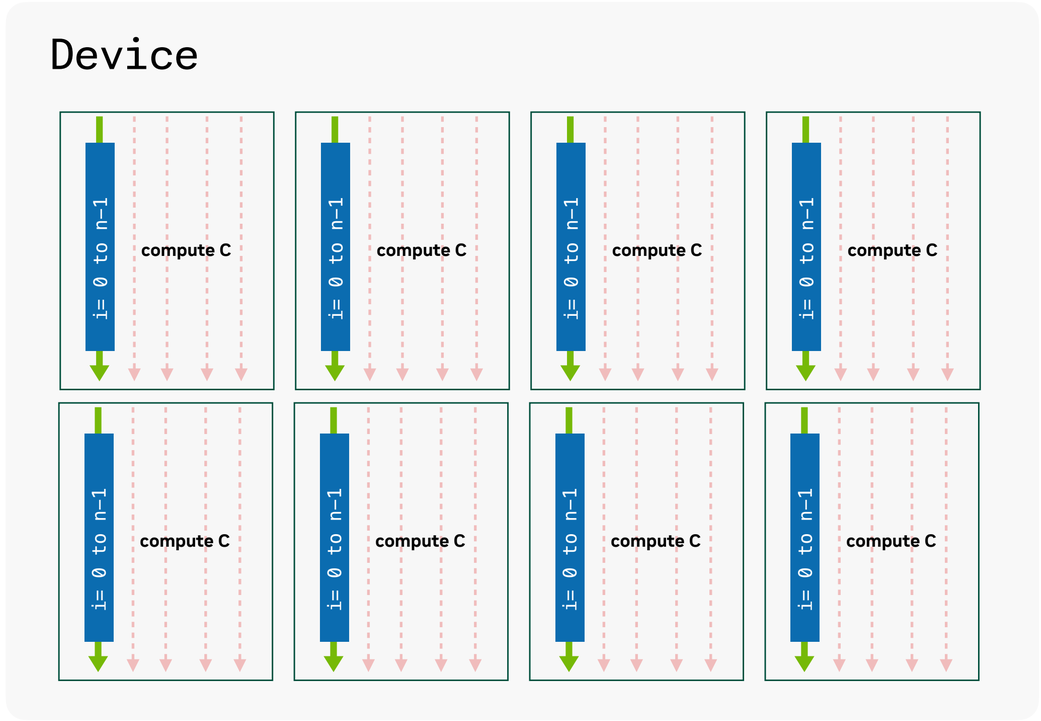
Parallelism identified by programer

Parallel | Kernels | loop

Parallel: a parallel region of code. The compiler generates a parallel kernel for that region. Each gang will execute the entire loop

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

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

  #pragma acc loop
  for (int i=0; i<N; I++) {d[I] = a*x[I] + y[I]}
}
```



Parallelism identified by programer

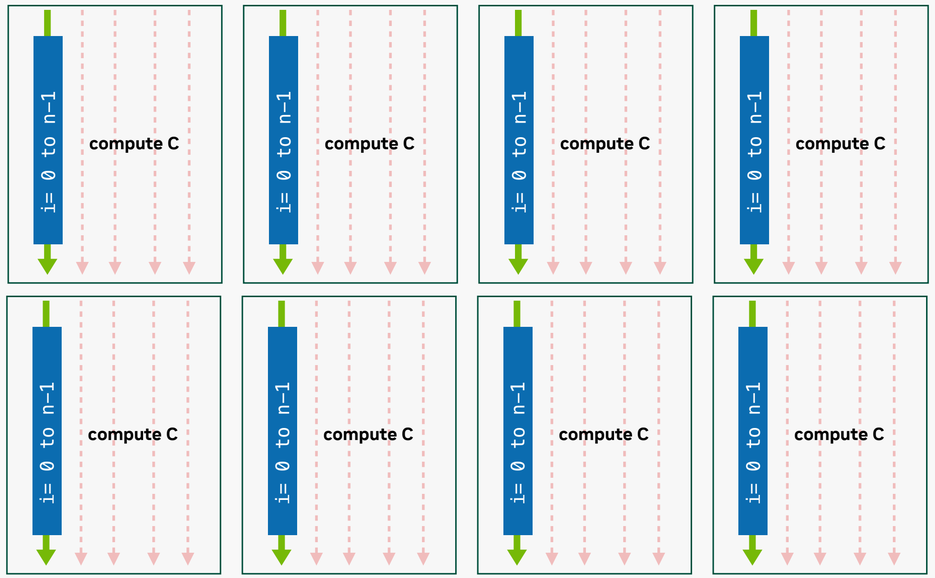
Parallel | Kernels | loop

Kernels: a parallel region of code. It allows the programmer to step back, and rely solely on the compiler

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

  for (int i=0; i<N; I++) {d[I] = a*x[I] + y[I]}

}
```

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

}
```

Device



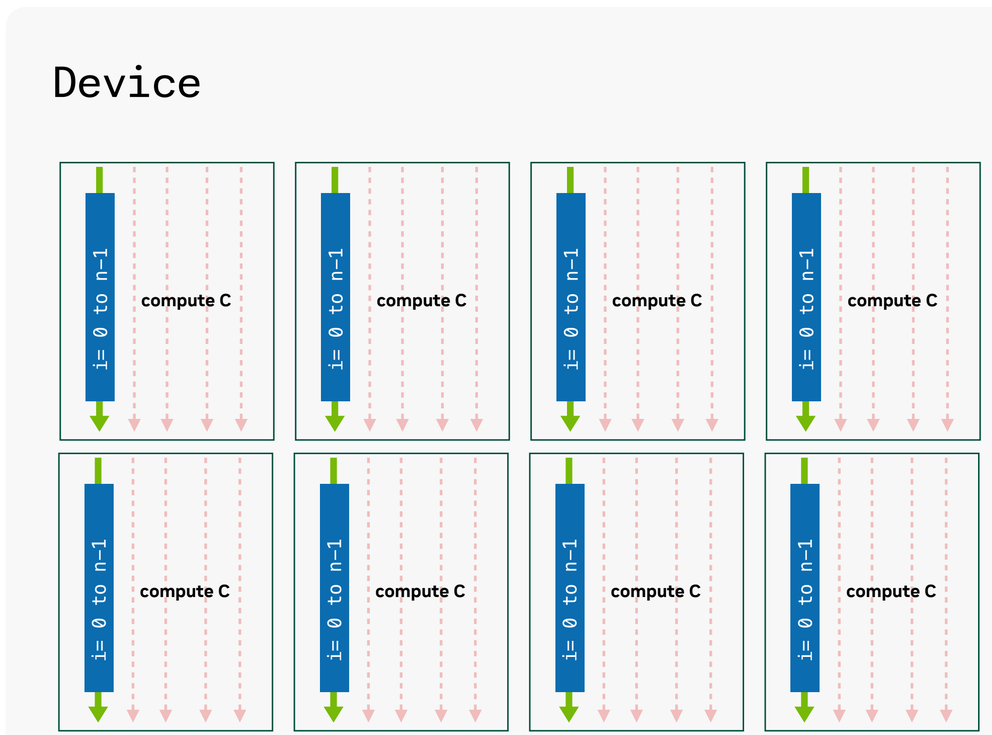
Parallelism identified by programer

Parallel | Kernels | loop

Loop: identifies a loop that should be distributed across threads. Parallel and loop are often placed together

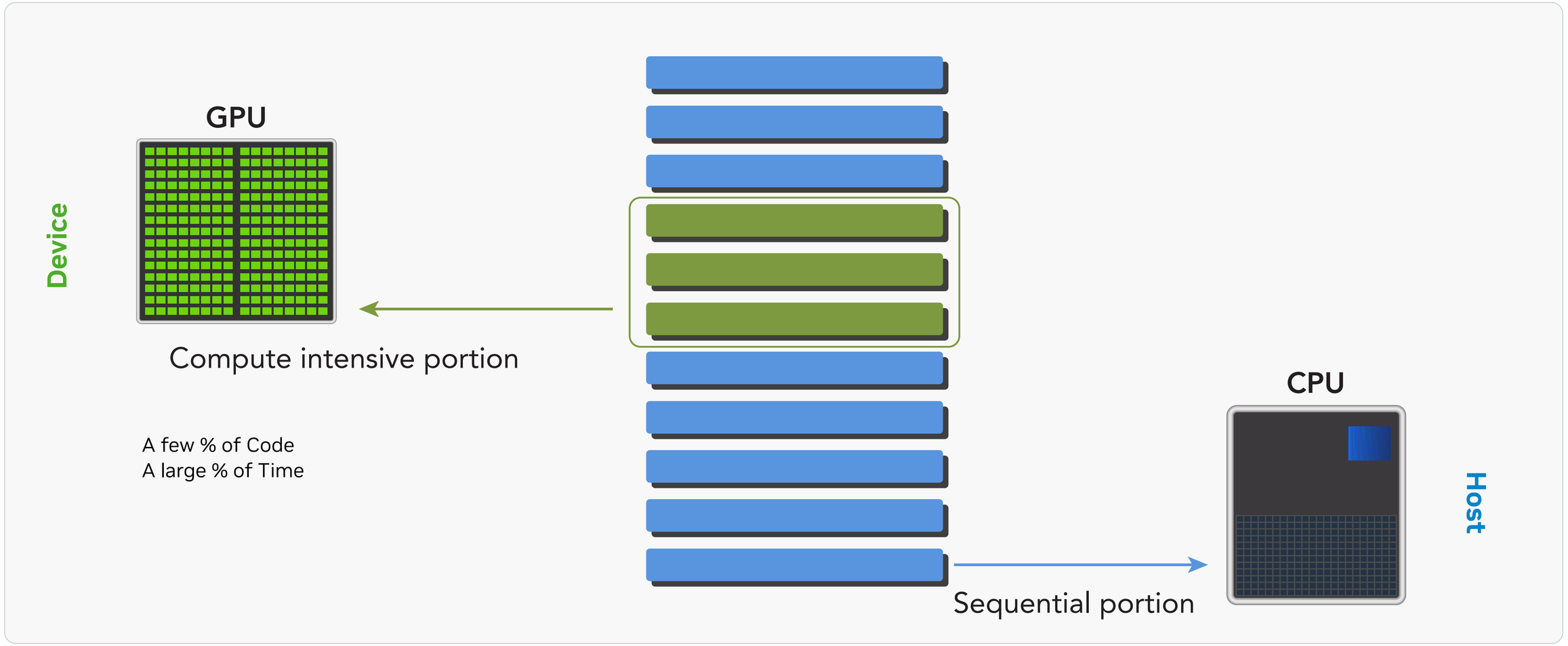
```
#pragma acc kernels
{
  #pragma acc loop independent
  for (int i=0; i<N; I++) {c[I] = a[i] + b[I]}

  #pragma acc loop independent
  for (int i=0; i<N; I++) {d[I] = a*x[I] + y[I]}

}
```

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

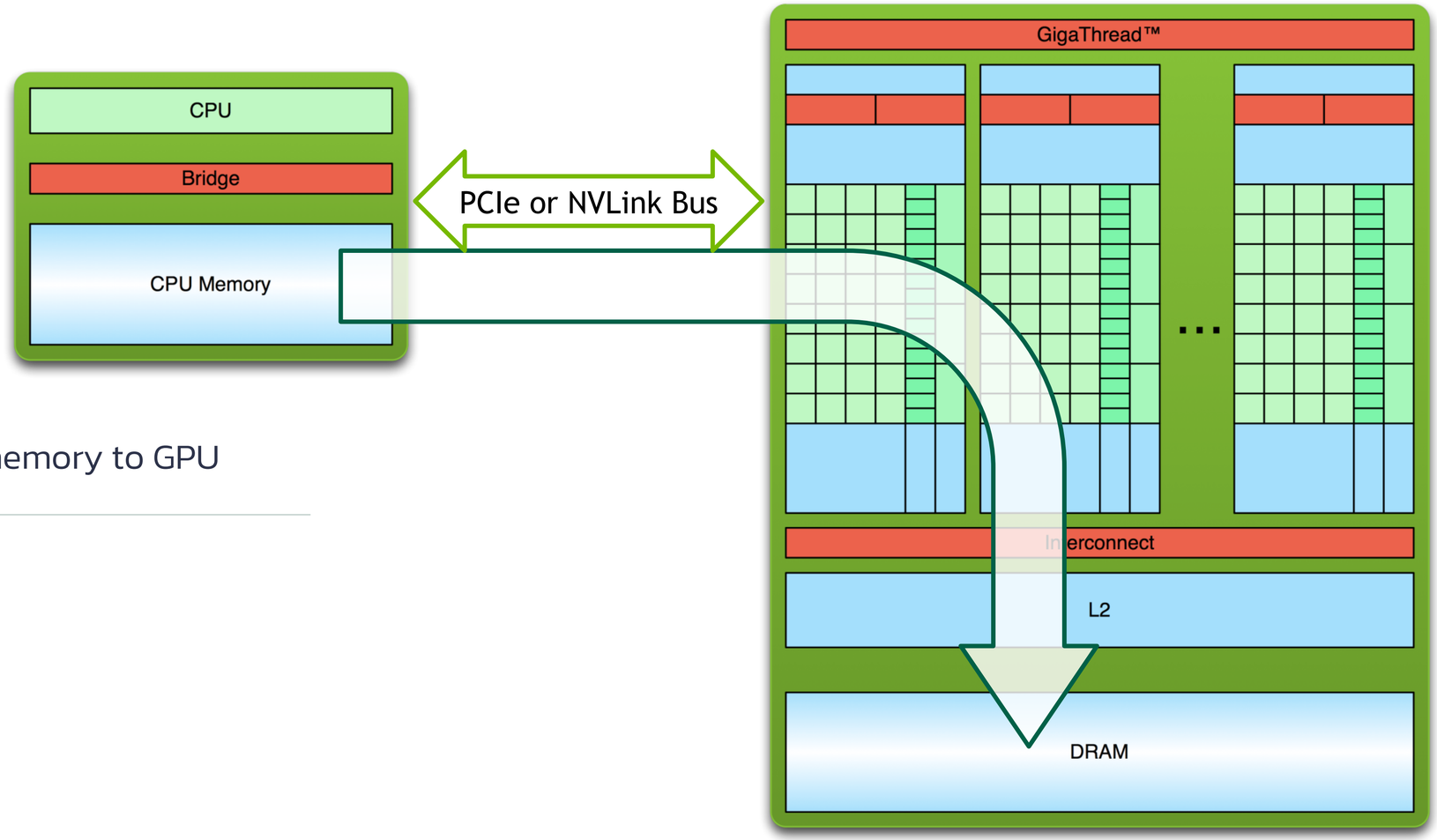


Accelerated Computing Fundamentals

What is heterogeneous programming?

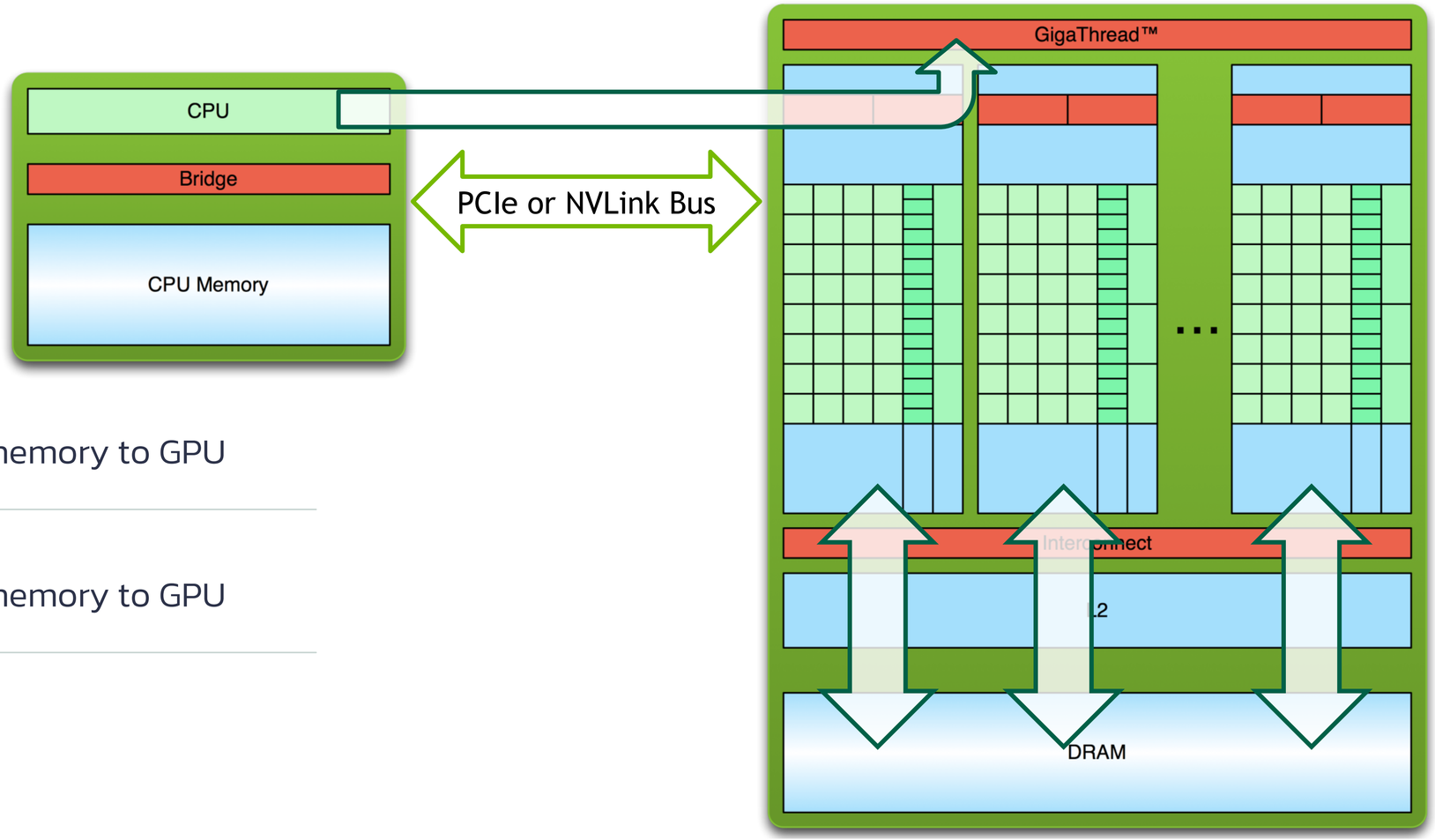


Three simple processing steps



1 Copy input data from CPU memory to GPU

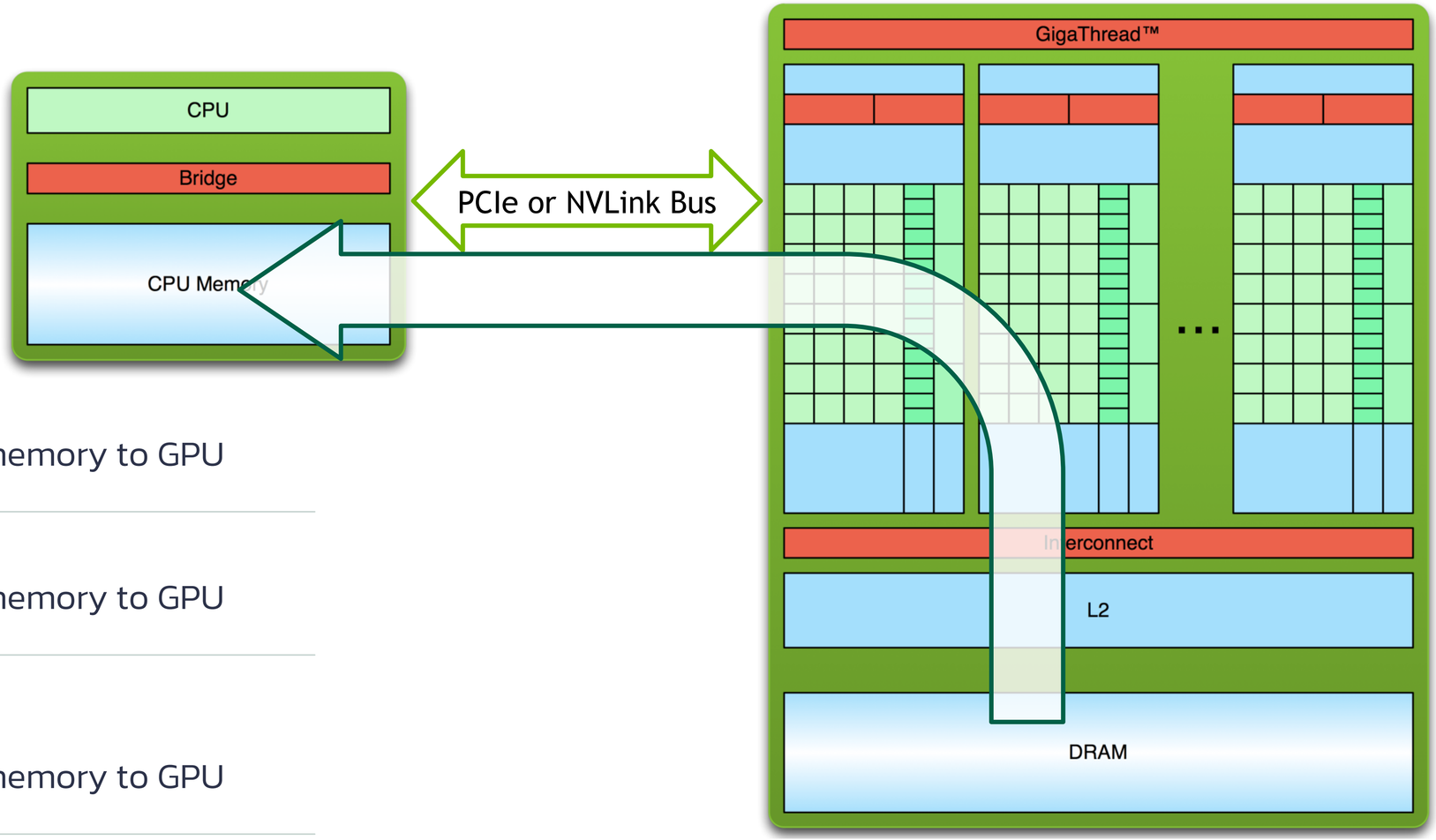
Three simple processing steps



1. Copy input data from CPU memory to GPU

2. Copy input data from CPU memory to GPU

Three simple processing steps



1. Copy input data from CPU memory to GPU
2. Copy input data from CPU memory to GPU
3. Copy input data from CPU memory to GPU

Data Management with OpenACC

1. Allocate ‘A’ on GPU
2. Copy From CPU to GPU
3. Copy back to GPU and deallocate A from GPU

```
int *A = (int*) malloc(N * sizeof(int));
#pragma acc parallel loop copy(A[0:N])
{
  for (int i=0; i<N; i++) A[i] = 0;
}
```

```
for (int i=0; i<N; I++) A[i] = 0;
#pragma acc parallel loop copy(A[0:N])
{
  for (int i=0; i<N; I++) A[i] = 1;
}
```

```
#pragma acc parallel loop copy(A[0:N], B[0:N])
{
  for (int i=0; i<N; I++) A[i] = B[I];
}
```

Data Management with OpenACC

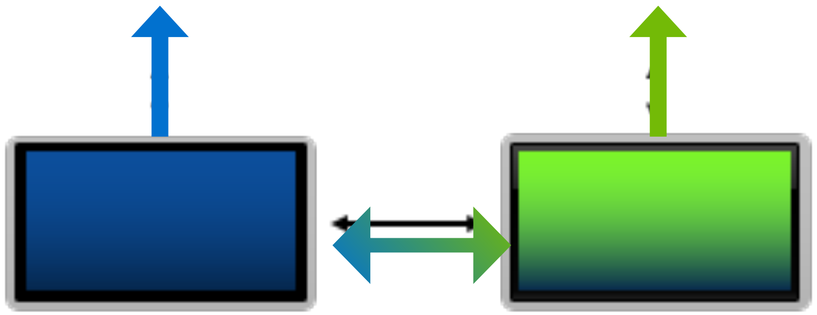
```
int *A = (int*) malloc(N * sizeof(int));
for (int i=0; i<N; I++)
A[i] = 0;
#pragma acc parallel loop
{
  for (int i=0; i<N; I++) A[i] = 1;
}
```

1. Allocate ‘A’ on GPU
2. Copy From CPU to GPU
3. Copy back to GPU and deallocate A from GPU

```
int *A = (int*) malloc(N * sizeof(int));
#pragma acc parallel loop copy(A[0:N]) copy(A[0:N])
{
  for (int i=0; i<N; i++)
     A[i] = 1;
     A[i] = B[i];
}
```

Managed Memory

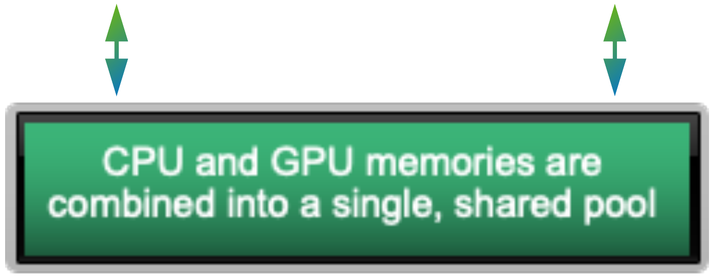
Without Managed memory



System memory GPU memory

```
$ nccx -gpu=cc80 -Minfo=acc -o binary Code.c
```

With Managed memory



Managed memory

```
$ nccx -gpu=cc80,managed -Minfo=acc -o binary Code.c
```

Profiling a GPU code

Nsight product family

Nsight System

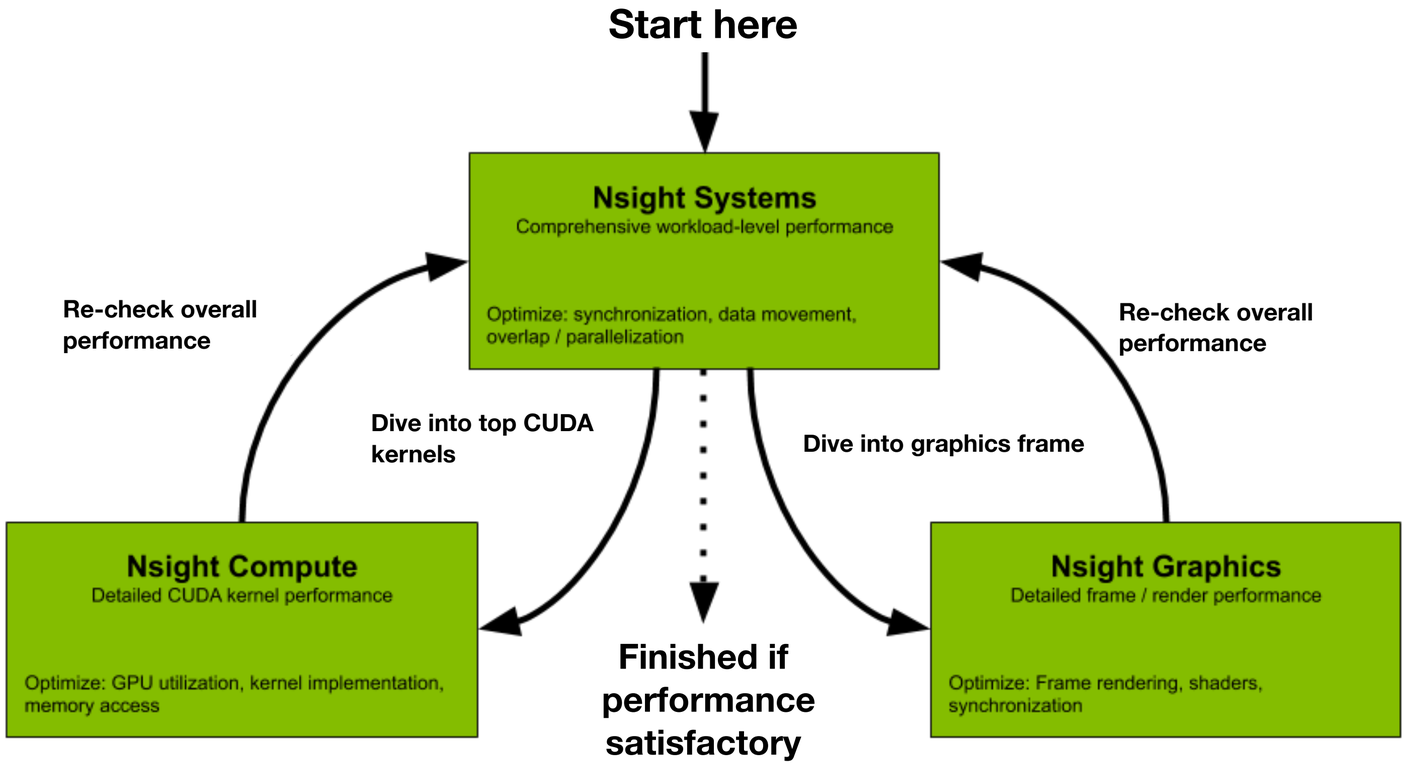
Analyze application algorithm system-wide

Nsight Compute

Debug/ Optimise CUDA kernel

Nsight Graphics

Debug/ Optimise graphics workloads



NSIGHT: Recording An Application Timeline

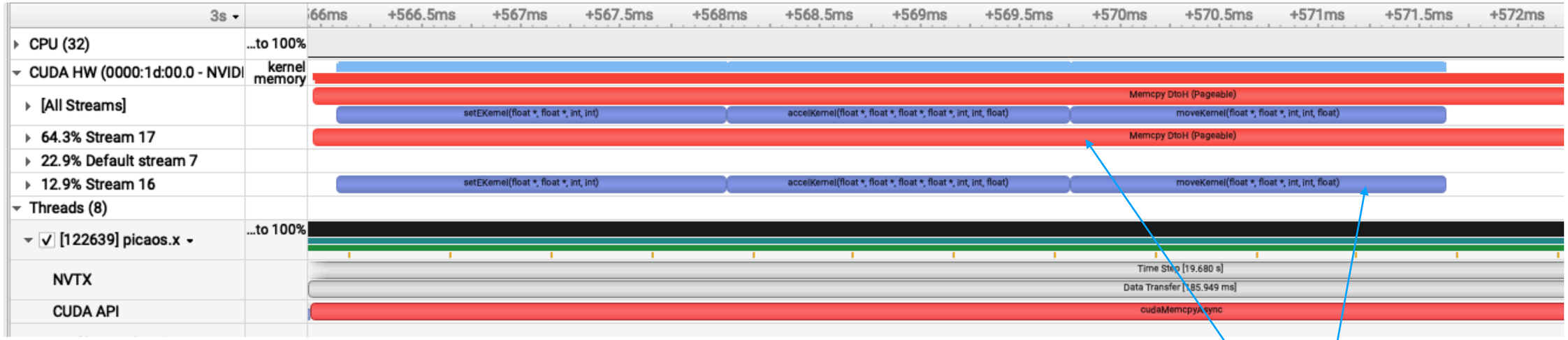
nsys profile -t **cuda,nvtx,mpi,openmp** --stats=true --force-overwrite true -o my_report ./myapp

- profile – start a profiling session
- -t: Selects the APIs to be traced (cuda, cublas, nvtx, mpi openmp and openacc in this example)
- —cuda-memory-usage = true or false
- --stats: if true, it generates summary of statistics after the collection
- --force-overwrite: if true, it overwrites the existing generated report
- -o – name for the intermediate result file, created at the end of the collection (.qdrep filename)

nsys --help or nsys [specific command] --help

Inspect results: Open the report file in the GUI
See also https://docs.nvidia.com/nsight-systems/

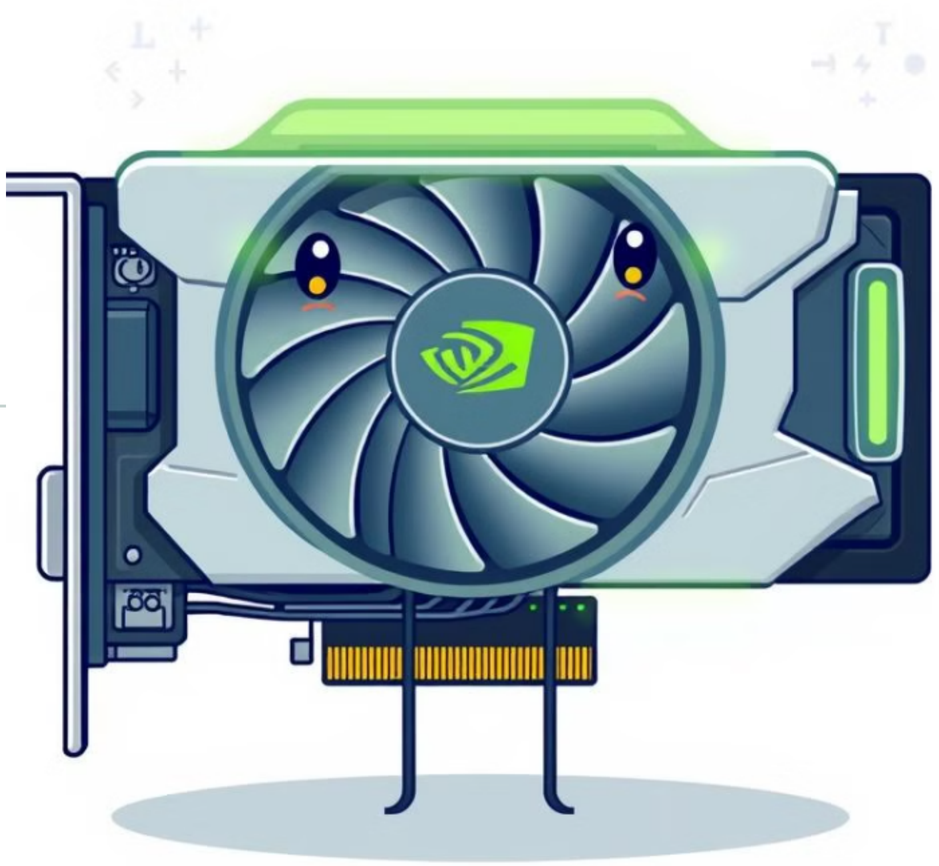
Nsight system report



Look at this pattern

1

Data Management with OpenACC



OpenACC Structured Data Directive

copyin(list) - Allocates memory on GPU and copies data from CPU(host) to GPU when entering a region

copyout(list) - Allocates memory on GPU and copies data to the CPU(host)host when exiting a region

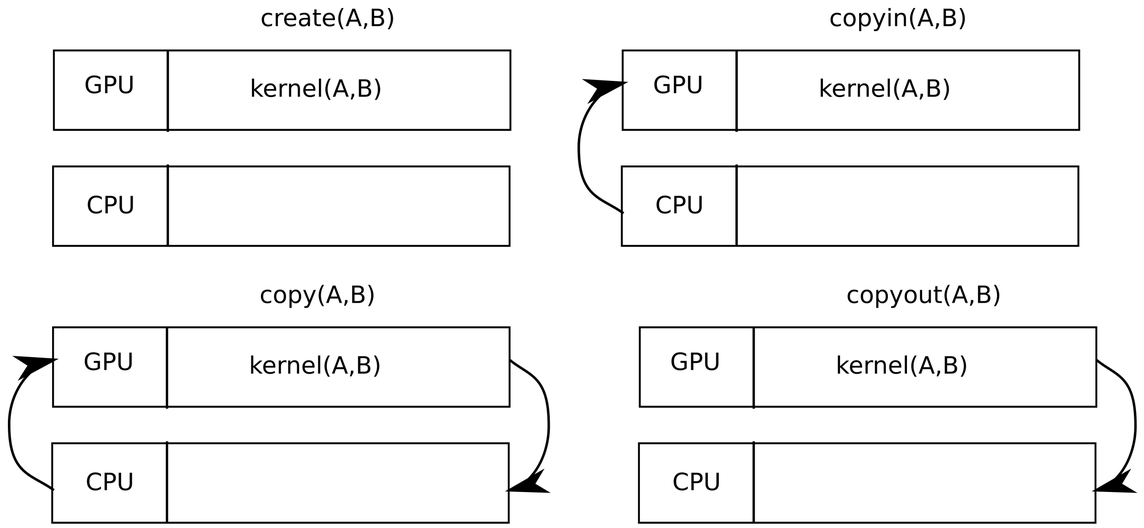
copy(list) - Allocates memory on GPU and copies data from CPU(host) to GPU when entering region and copies data to the CPU(host) when exiting a region

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

delete(list) - Deallocate memory on the GPU without copying

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

and **present_or_copy[inlout], present_or_create, deviceptr.**



Encompassing Multiple Compute Regions

```
#pragma acc data copyin(A[0:N], B[0:N]) create(C[0:N])
{

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

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

}
```

```
void copy(int *A, int *B, int N)
{
#pragma acc parallel loop copyout(A[0:N]) copyin(B[0:N])
    for( int i = 0; i < N; i++ )
    {
        A[i] = B[i];
    }
}
```

```
#pragma acc data copyout(A[0:N],B[0:N]) copyin(C[0:N])
{
    copy(A, C, N);

    copy(A, B, N);
}
```

Array Shaping

Compiler sometimes cannot determine size of arrays

- Must specify explicitly start/end point
- Memory only exits within the data region
- Must be within a single function

C/C++

```
#pragma acc data copyin(a[0:nelem]) copyout(b[s/4:3*s/4])
```

Fortran

```
!$acc data copyin(a(1:end)) copyout(b(s/4:3*s/4))
```

- Fortran uses *start:end* and C uses *start:count*
- Data clauses can be used on data, kernels or parallel

OpenACC Unstructured Data Directive

There are two unstructured data directives:

Enter data: Handles device memory allocation, and copies from the Host to the Device. The two clauses that you may use with `enter data` are `create` for device memory allocation, and `copyin` for allocation, and memory copy.

Exit data: Handles device memory deallocation, and copies from the Device to the Host. The two clauses that you may use with `exit data` are `delete` for device memory deallocation, and `copyout` for deallocation, and memory copy.

The unstructured data directives do not mark a "data region", because you are able to have multiple `enter data` and `exit data` directives in your code. It is better to think of them purely as memory allocation and deallocation.

```
int* allocate(int size)
{
    int *ptr = (int*) malloc(size * sizeof(int));
    #pragma acc enter data create(ptr[0:size])
    return ptr;
}

void deallocate(int *ptr)
{
    #pragma acc exit data delete(ptr)
    free(ptr);
}

int main()
{
    int *ptr = allocate(100);

    #pragma acc parallel loop
    for( int i = 0; i < 100; i++ )
    {
        ptr[i] = 0;
    }

    deallocate(ptr);
}
```

OpenACC Update Directive

self: The self clause will transfer data from the device to the host (GPU to CPU)

device: The device clause will transfer data from the host to the device (CPU to GPU)

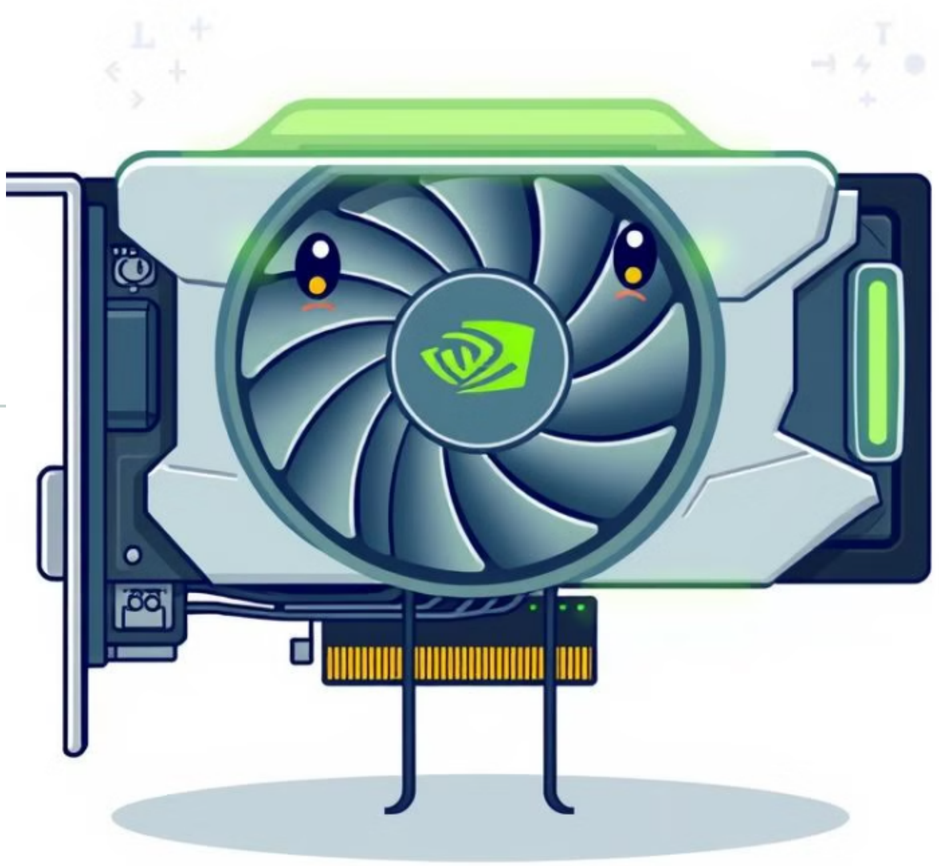
The syntax would look like:

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

```
#pragma acc data copyin( A[:m*n],Anew[:m*n] )
{
  while ( error > tol && iter < iter_max ) {
    error = calcNext(A, Anew, m, n);
    swap(A, Anew, m, n);

    if(iter % 100 == 0) {

    printf("%5d, %0.6f\n", iter, error);

    #pragma acc update self(A[0:m*n])

     for( int i = 0; i < n; i++ ) {
      for( int j = 0; j < m; j++ ) {
            printf("%0.2f ", A[i+j*m]);
         }
            printf("\n");
       }
      }
       iter++;
    }
}
```

2

OpenACC Loop Optimization



Independent Clause

Independent Clause: A way for programmer to guarantee to the compiler that a loop is parallelizable

- Overrides compiler analysis for dependence, could force the compiler to parallelise a non-parallel loop
- In a kernel construct the loop independent tells to the compiler that the items in the loop are not containing any dependence on each other
- When using parallel directive, the independent clause is implied

```
#pragma acc kernels loop
independent
for ( int i = 0; i < n; i++)
          < Parallel Loop >
```

```
#pragma acc kernels loop
for ( int i = 0; i < n; i++)
          < Parallel Loop >

#pragma acc independent
for ( int i = 0; i < n; i++)
          < Parallel Loop >
```

```
#pragma acc kernels
#pragma acc loop independent
for ( int i = 0; i < n; i++)
          c[i] = 2.*c[m+i];
```

m>n

Auto Clause: more-or-less the opposite of the Independent

Auto Clause

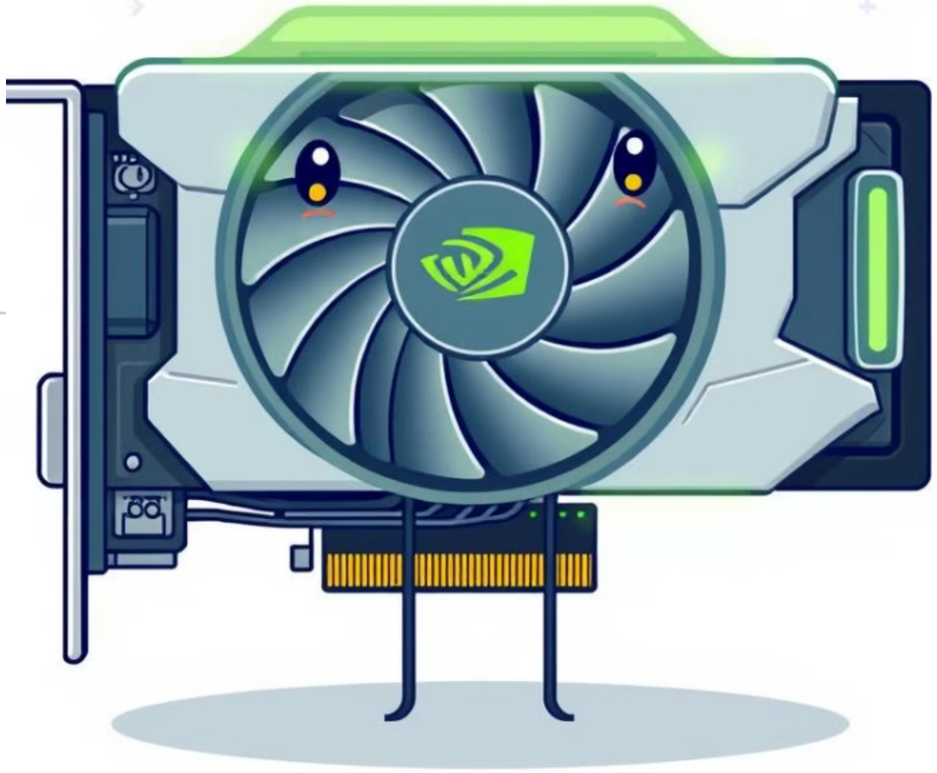
Auto Clause: more-or-less the opposite of the Independent

- the compiler will trust anything that the programmer decides
- the compiler will double check the loops, and decide whether or not to parallelize them
- You do not need to include the auto clause when using kernels directives
- However, the auto clause can be very useful when the parallel directives

```
#pragma acc kernels loop auto
for ( int i = 0; i < n; i++)
          < Parallel Loop >
```

2

Loop Correctness



SEQ Clause

SEQ Clause: short for sequential

- the compiler will tell to run the loop
- Compiler will parallelise the outer loop across the parallel threads, but each thread will run the inner-most loop sequentially
- It may automatically apply the seq clause to loops that have too any dimensions

```
#pragma acc parallel loop
  for (int i = 0; i < size; I++)

    #pragma acc loop
    for (int j = 0; j < size; j++)

      #pragma acc loop seq
      for (int k = 0; k < size; k++)

        c[i][j] += a[I][j] * b[I][j];
```

Private and Firstprivate Clause

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 comma-separated list

The **Firstivate Clause:** like private except

- The private values are initialised to the same value used on the host. Private values are uninitizlised

```
double tmp[3]
 #pragma acc kernels loop privarte(tmp[0:3]
  for (int i = 0; i < size; i++)
    tmp[0] += <value>;
    tmp[1] += <value>;
    tmp[2] += <value>;
#pragma acc parallel loop
  for (int j = 0; j < size; I++)
      array[I][j] = tmp[0] + tmp[1] + tmp[2]
```

tmp array is private to each iteration of the outer loop

Shared among the threads in the inner loop

Scalars and Private clause

By default, scalars are **firstprivate** when used in a parallel region and **private** when used in a kernels region

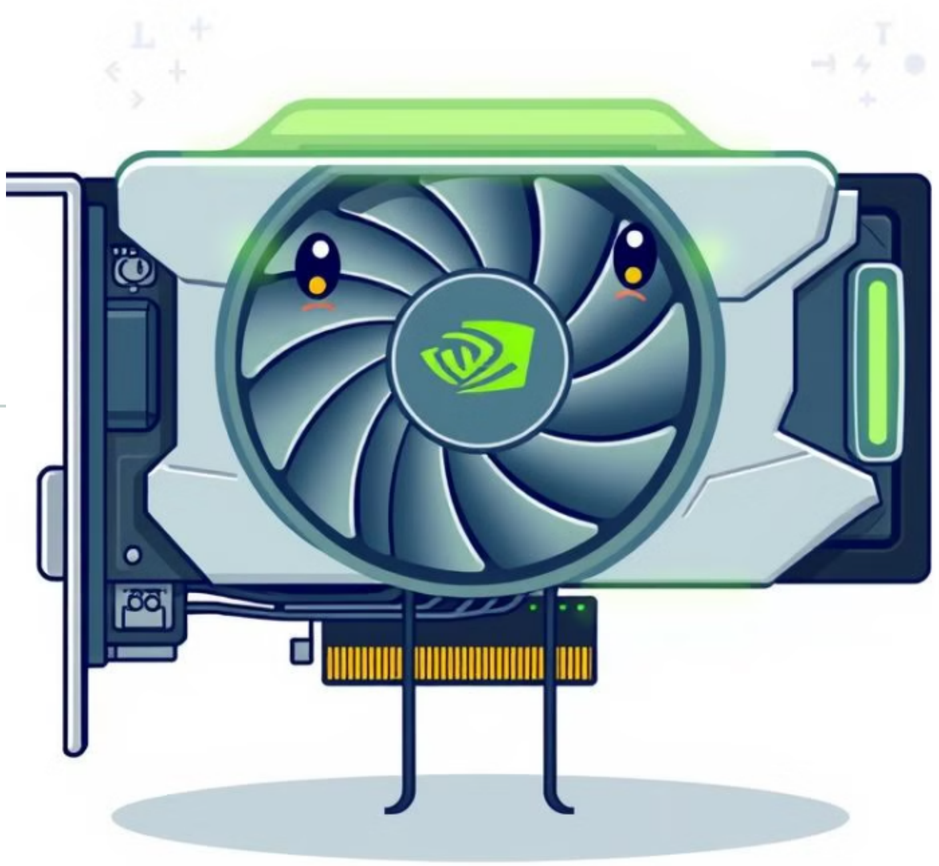
Except in some cases, scalars do not need to be added to a private clause. These cases may include but are not limited to:

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

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

2

Loop Optimization



Collapse Clause

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

```
#pragma acc parallel loop collapse(2)
for ( int i = 0; i < N; i++)
    for ( int j = 0; j < N; j++)
       structured-block
```

The Tile clause

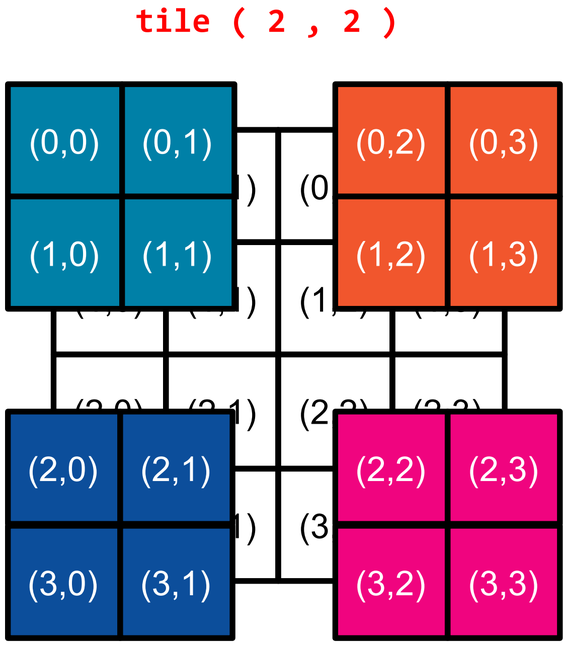
tile(x, y, z . . . .)

- Breaks multidimensional loop into “blocks ”
- Can increase data locality in some codes
- Will be able to execute multiple tiles simultaneously

```
#pragma acc kernels loop tile(32,32)
for ( int i = 0; i < 128; i++)
    for ( int i = 0; i < 128; i++)
     structured-block
```

Similar to the `collapse` clause, the inner loops should not have the `loop` directive.

```
#pragma acc kernels loop tile(32,32)
for ( int i = 0; i < 128; I++)
     #pragma acc loop
    for ( int i = 0; i < 128; i++)
```



Reduction Clause

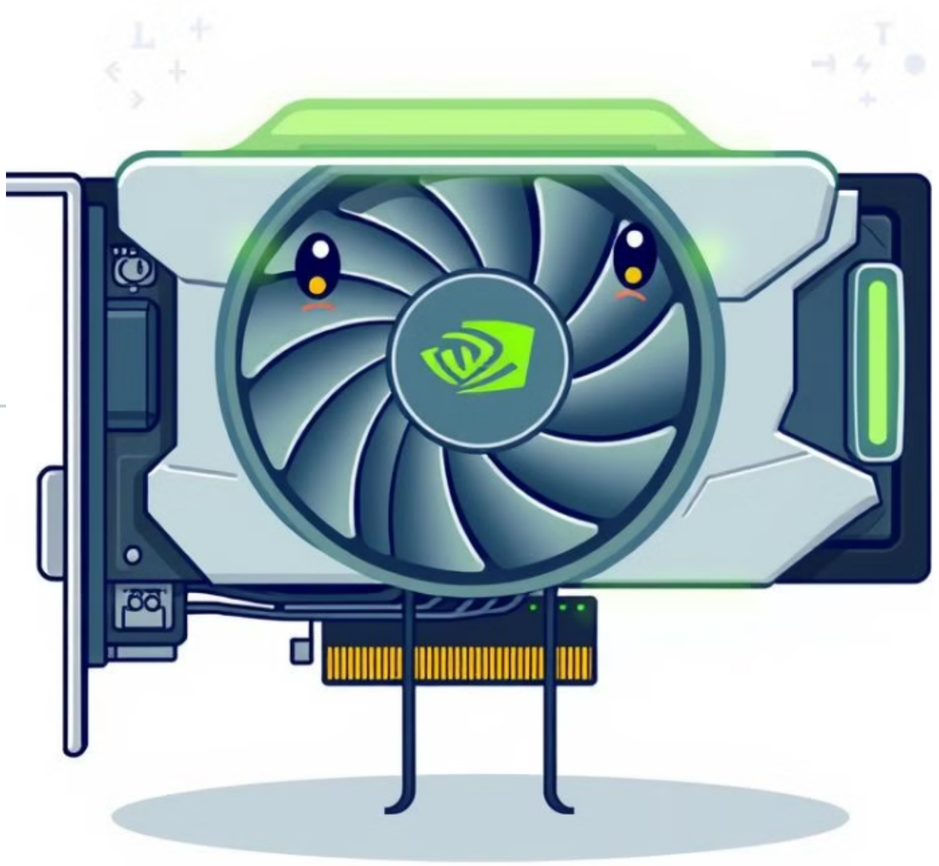
- Takes many values and “reduces” to a single value
- Each thread calculates its part
- Perform a partial reduction on the loop iterations the they compute
- After the loop, the compiler will perform a final reduction to produce a single result using the specified operation

```
for (int i = 0; i < size; i++)
   for (int j = 0; j < size; j++)
   #pragma acc parallel loop reduction(+:tmp)
        for (int k = 0; k < size; j++)
            tmp += a[i][j] + b[i][j];
            c[i][j] = tmp;
```

Operator	Example	Description
+	reduction(+:sum)	Mathematical summation
*	reduction(*:product)	Mathematical product
max	reduction(max:maximum)	Maximum value
min	reduction(min:minimum)	Minimum value
&	reduction(&:val)	Bitwise AND
|	reduction(|:val)	Bitwise OR
&&	reduction(&&:bool)	Logical AND
||	reduction(||:bool)	Logical OR

3

GPU hardware hierarchy



A simplified representation of a NVIDIA GPU-architecture

Peak performance NVIDIA A100

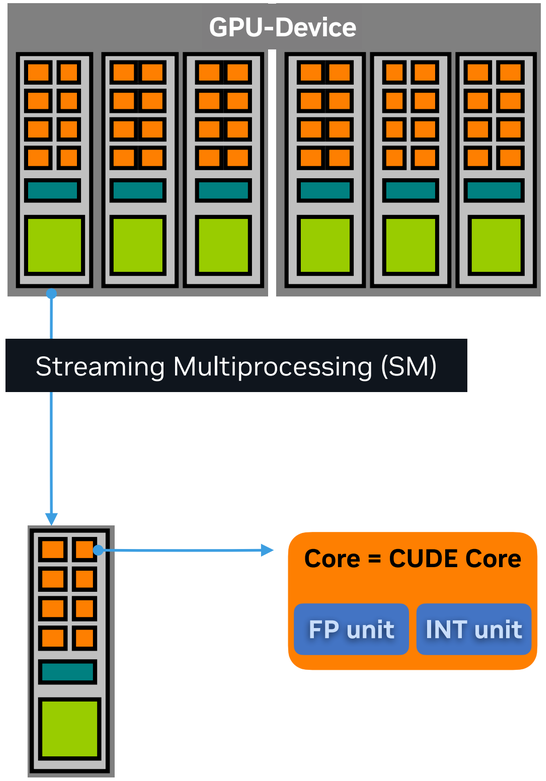
FLOPS = Clock speed × cores × FLOP/cycle

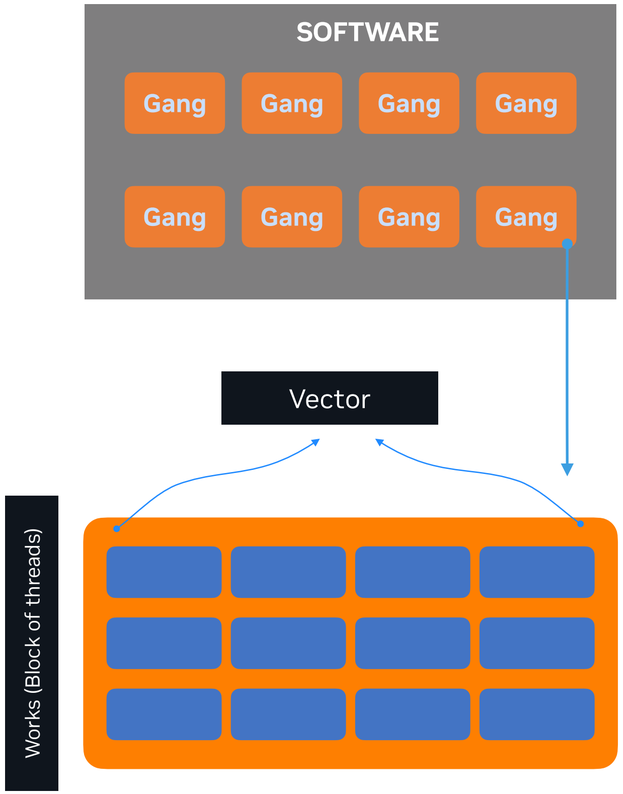
FLOPS <=> FP64 or FP32 or ...

Clock speed (or GPU Boost Clock) = 1.41 GHz

Total FLOPS = 250.0 TFLOPS

(Thread ∈ Block ∈ Grid)





Gang Worker Vector DEMYSTIFIED

OpenACC
More Science, Less Programming

NVIDIA

Gang Worker Vector DEMYSTIFIED

OpenACC More Science, Less Programming

NVIDIA

How much work 1 worker can do is limited by his speed.

A single worker can only move so fast.

Gang Worker Vector DEMYSTIFIED

OpenACC More Science, Less Programming

NVIDIA

Even if we increase the size of his roller, he can only paint so fast.

We need more workers!

Gang Worker Vector DEMYSTIFIED

OpenACC More Science, Less Programming

NVIDIA

Gang Worker Vector DEMYSTIFIED

By organizing our workers into groups (gangs), they can effectively work together within a floor.

Groups (gangs) on different floors can operate independently.

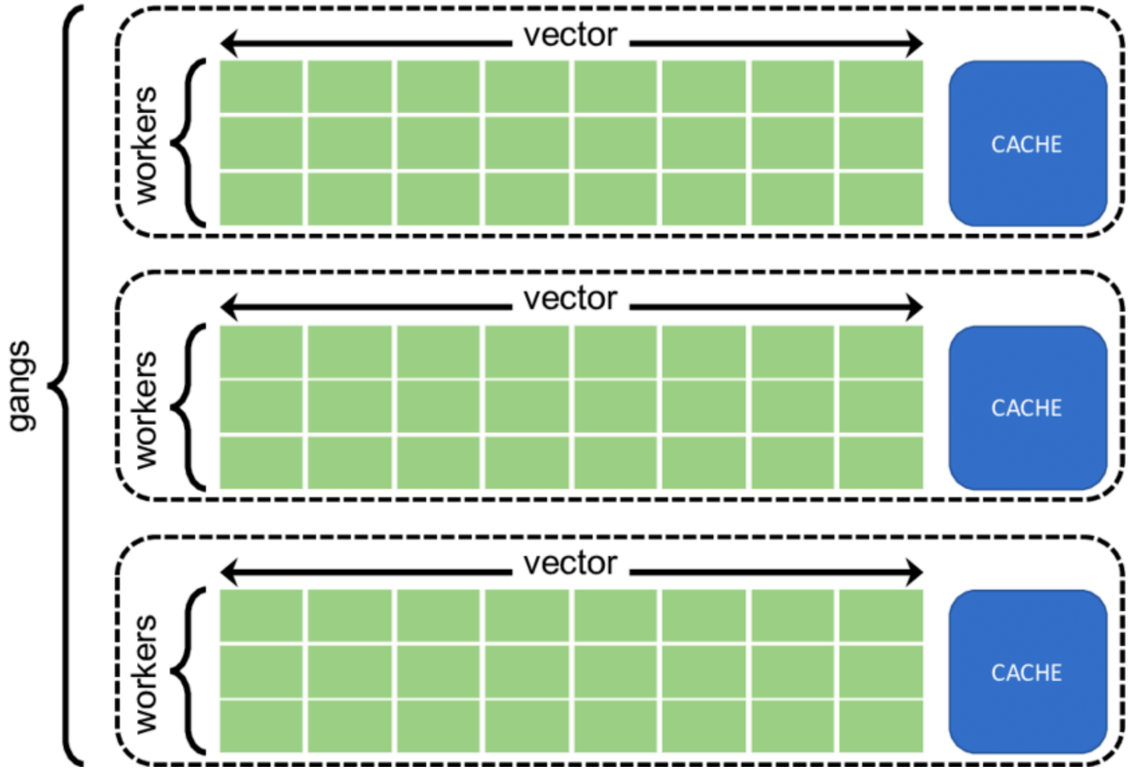
Since gangs operate independently, we can use as many or few as we need.

Gang Worker Vector DEMYSTIFIED

Our painter is like an OpenACC ***worker***, he can only do so much.

His roller is like a ***vector***, he can move faster by covering more wall at once.

Eventually we need more workers, which can be organized into ***gangs*** to get more done.



OpenACC More Science, Less Programming

NVIDIA

Gang Worker Vector

Gang

- Multiple gangs will be generated, and loops iterations will be spread across the gangs
- Gangs are independent of each other
- There is no way to for the programmer to know exactly how many gangs are running at a given time

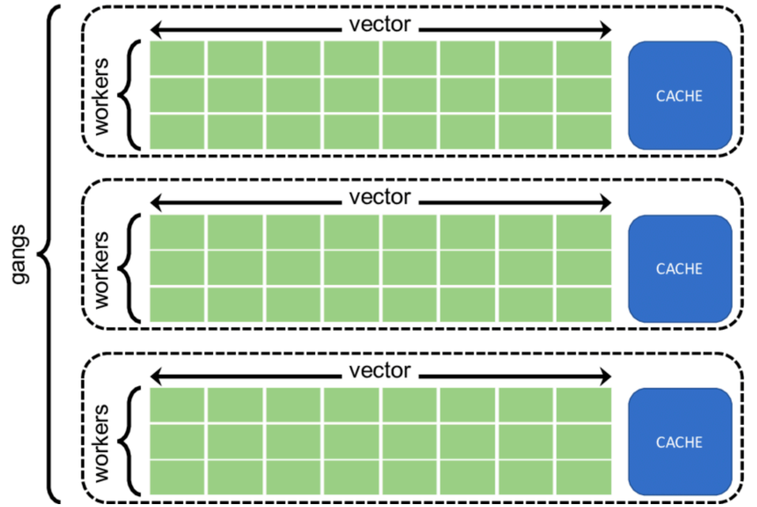
Worker

- To have **multiple vectors** within a gang
- Splits up one large vector into multiple smaller vectors
- Intermediate level between the low-level parallelism implemented in vector and group of threads
- useful when our inner parallel loops are very small, and will not benefit from having a large vector

Vector parallelism:

- Lowest level of parallelism
- Every gang will have at least 1 vector
- Threads work in lockstep (SIMD/SIMT parallelism)

```
#pragma acc loop gang
for ( int i = 0; i < N; i++)
 #pragma acc loop worker
  for ( int j = 0; j < N; j++)
  #pragma acc loop vector
   for ( int k = 0; k < N; k++)
          structured-block
```



Controlling the size of Gang, Worker and Vectors

The compiler will choose a number of gangs, workers, and a vector length for you, but you can change it with clauses

num_gangs(N)

- Generate N gangs for this parallel region

num_workers(M)

- Generate N gangs for this parallel region

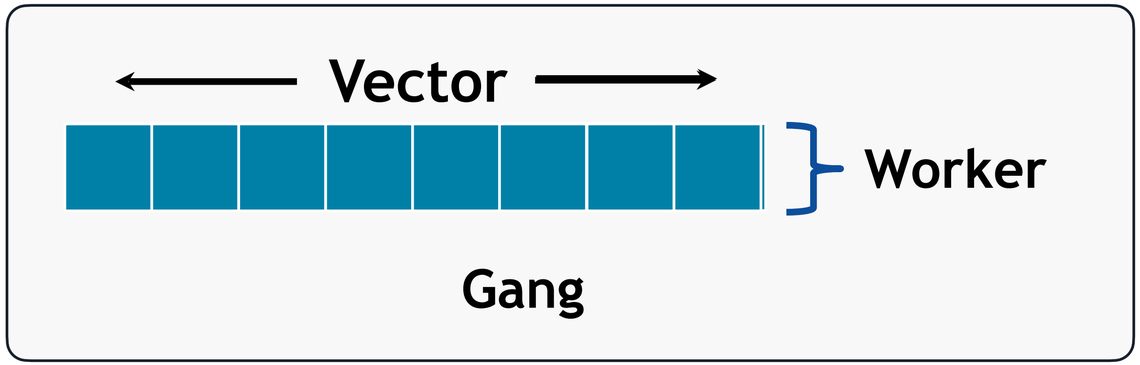
vector_length((P):

- Use a vector length of P for this parallel region

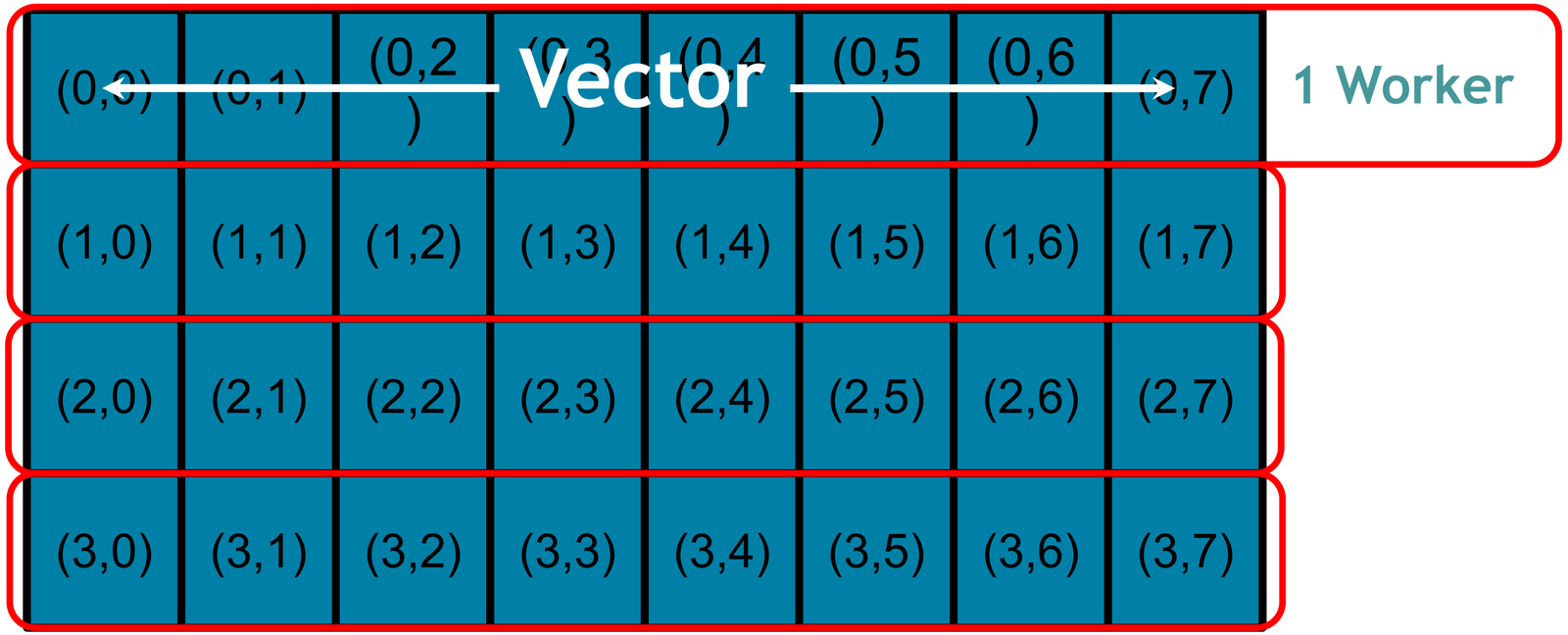
```
#pragma acc paralle nums_gangs(2) num_workers(4) vector_length(32)
 {
  #pragma acc loop worker
   for ( int i = 0; i < N; i++)
     #pragma acc loop vector
     for ( int j = 0; j < N; j++)
            structured-block
 }
```

Rule of 32: general rule of thumb for programming for NVIDIA GPUs is to always ensure that your vector length is a multiple of 32 (which means 32, 64, 96, 128, … 512, … 1024… etc.)

Gang Worker Vector DEMYSTIFIED



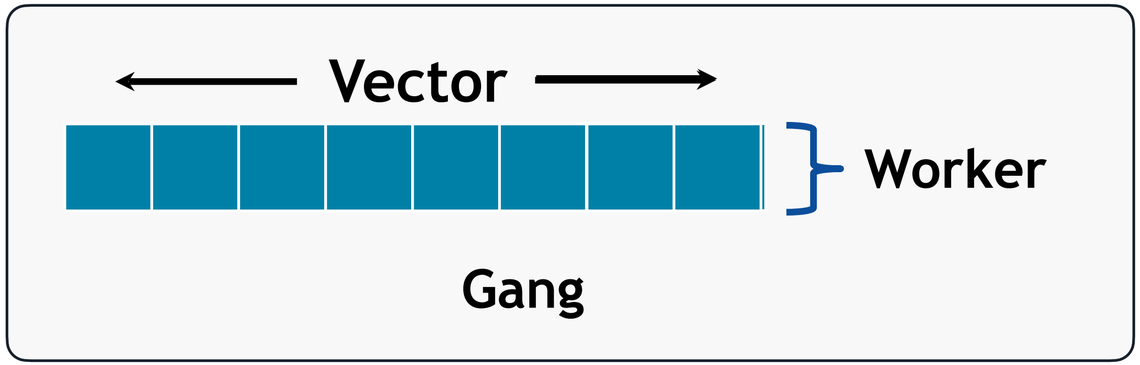

```
#pragma acc kernels loop gang worker(1)
for(int x = 0; x < 4; x++){
   #pragma acc loop vector(8)
    for(int y = 0; y < 8; y++){
        array[x][y]++;
     }
}
```



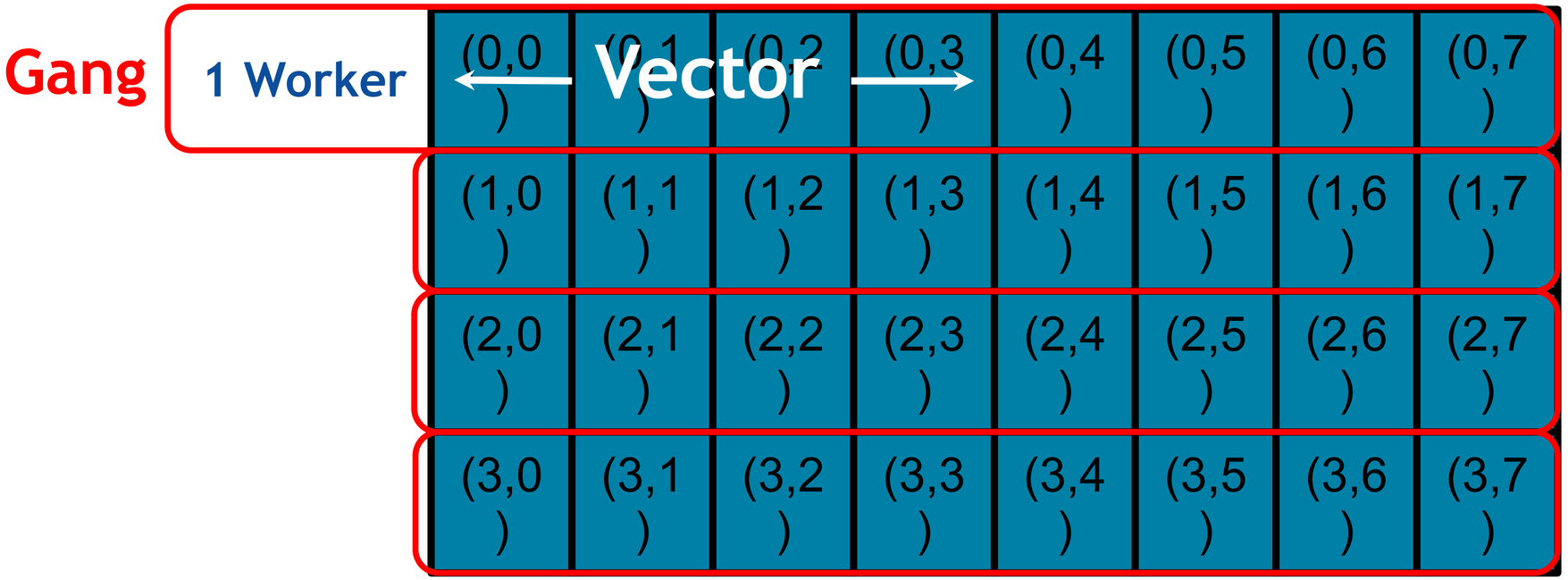
Gang

- 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 DEMYSTIFIED

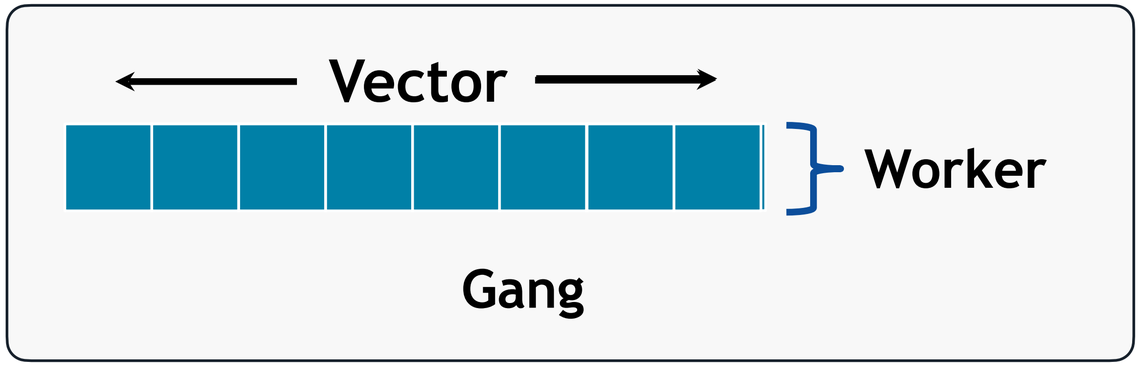



```
#pragma acc kernels loop gang worker(1)
for(int x = 0; x < 4; x++){
   #pragma acc loop vector(4)
    for(int y = 0; y < 8; y++){
        array[x][y]++;
     }
}
```

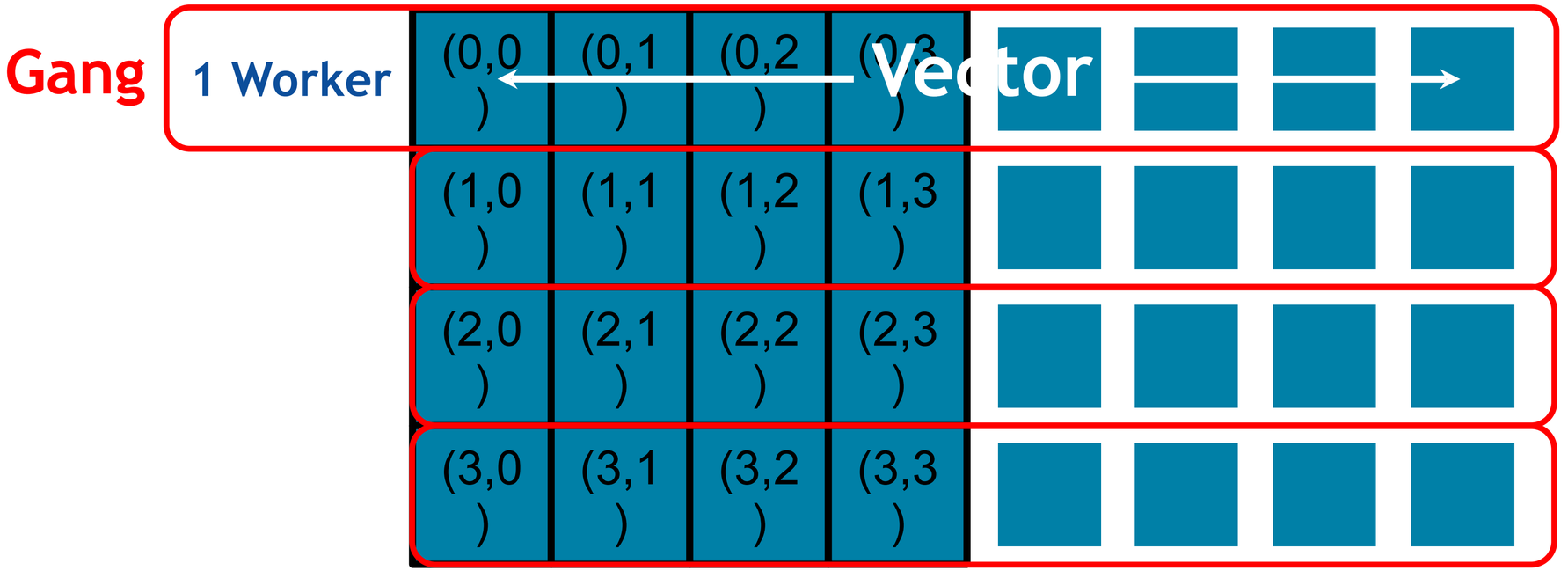


- We are still generating 4 gangs, but now each vector is computing two loop iterations
- If we wanted to generate **more gangs**, we would need to increase the size of the outer-loop

Gang Worker Vector DEMYSTIFIED

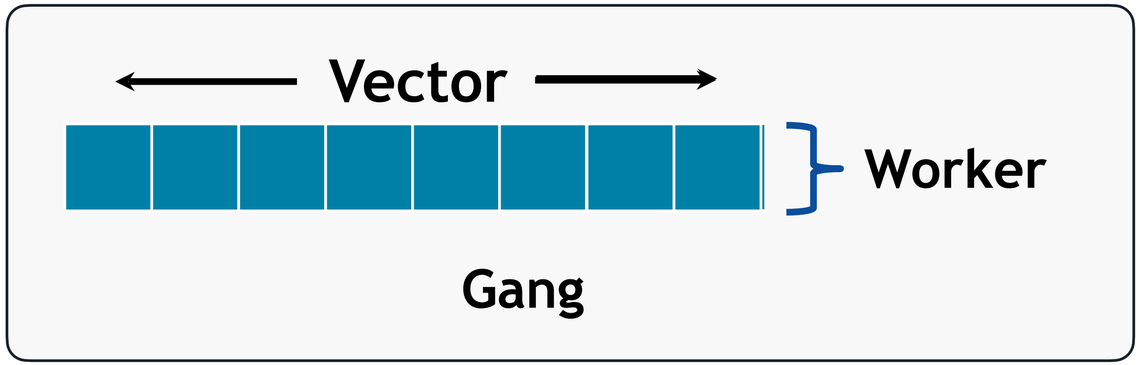



```
#pragma acc kernels loop gang worker(1)
for(int x = 0; x < 4; x++){
   #pragma acc loop vector(8)
    for(int y = 0; y < 4; y++){
        array[x][y]++;
     }
}
```



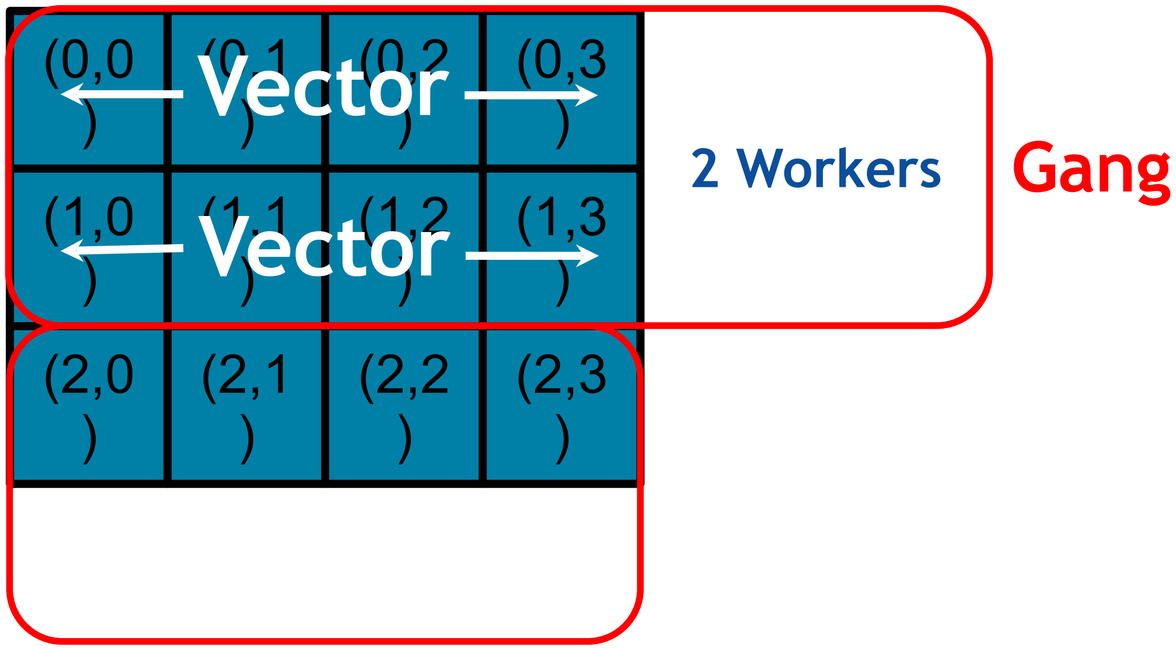
- We can see that our vector length is **much larger** than our inner-loop
- We are **wasting** half of our vector, meaning our code is performing half as well as it could

Gang Worker Vector DEMYSTIFIED



We can fix this by **breaking our vector** up among **2 workers**

```
#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