Introduction to OpenACC Programming

Dr. Ezhilmathi Krishnasamy

March 25, 2025

Outline

Outline for section 1

Objectives

- Understand the OpenACC programming model.
- Learn how to use various directives from OpenACC to parallelize code, including:
  - Compute constructs
  - Loop constructs
  - Data clauses
- Implement OpenACC parallel strategies in C/C++ and FORTRAN programming languages.
- Explore simple mathematical examples to enhance understanding of the OpenACC programming model.
- Finally, learn how to run these examples on the MeluXina Supercomputer:
  - Both interactively and using a batch job script.

Prerequisites

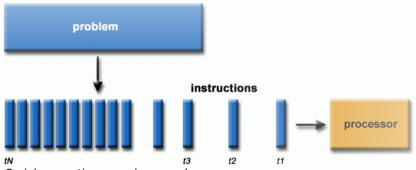
- Proficiency in C/C++ or FORTRAN languages.
- Familiarity with OpenMP or basic parallel programming concepts is beneficial but not required.

Outline for section 2

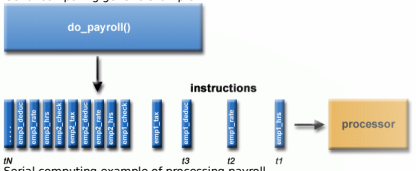
Serial Programming vs. Parallel Programming

- **Serial Programming:**
  - The problem is divided into a sequential series of instructions.
  - Instructions are executed one at a time in a single thread or processor.
  - Only one instruction can be processed simultaneously.
- **Parallel Programming:**
  - The problem is divided into discrete parts that can be solved concurrently.
  - Each part consists of its own set of instructions.
  - Instructions from different parts are executed on separate threads or processors.
  - Requires control and coordination to manage parallel execution effectively.
- **Hardware:**
  - CPU, GPU, and other parallel processors are capable of performing parallel computing.

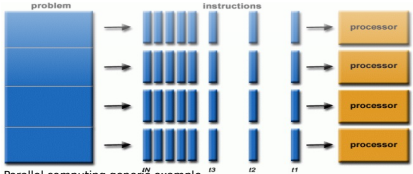
Serial Programming vs. Parallel Programming



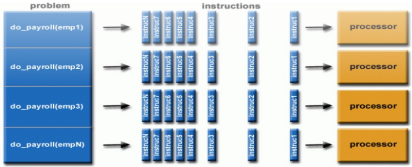
Serial computing generic example



Serial computing example of processing payroll



Parallel computing generic example



Parallel computing example of processing payroll

Source:HPC LLNL

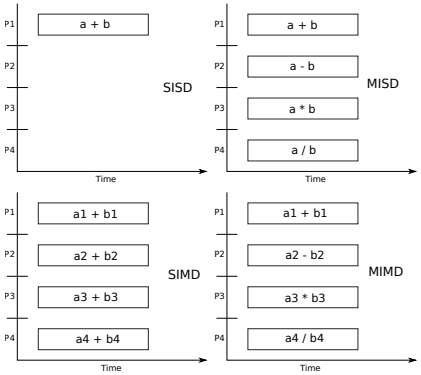
Outline for section 3

Classification of Parallel Computer Architecture

- **Control Structure:** Classified based on instruction and data streams.
  - **SISD:** Single Instruction Stream, Single Data Stream
  - **SIMD:** Single Instruction Stream, Multiple Data Streams
  - **MISD:** Multiple Instruction Streams, Single Data Stream
  - **MIMD:** Multiple Instruction Streams, Multiple Data Streams
- **Memory Organization:** Distinction between Shared Memory and Distributed Memory
- **Network Topology:** Examples include 3D Grid and Tree structures

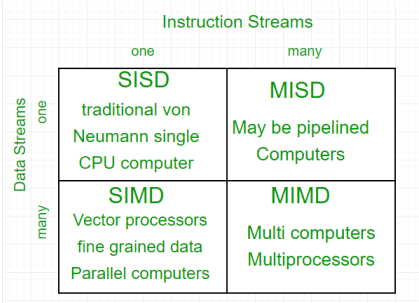
Control Structures - Flynn's Taxonomy

- **SISD (Single Instruction Stream, Single Data Stream)**: Represents sequential execution typical of conventional single processor systems based on the von Neumann architecture.
- **SIMD (Single Instruction Stream, Multiple Data Streams)**: Utilizes array computers and traditional vector processors, allowing instructions to be executed simultaneously in parallel or through pipelining.



Control Structures - Flynn's Taxonomy

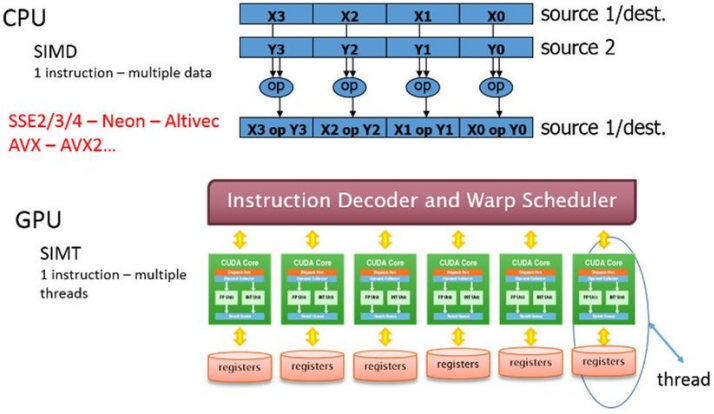
- **MISD (Multiple Instruction Streams, Single Data Stream)**: A rarely employed architecture with limited practical applications.
- **MIMD (Multiple Instruction Streams, Multiple Data Streams)**: Supports both shared and distributed memory architectures, enabling more complex multitasking and parallel processing capabilities.



Outline for section 4

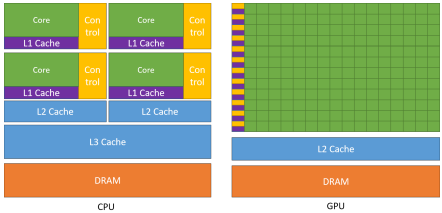
GPU Architecture

GPUs utilize Single Instruction Multiple Threads (SIMT)

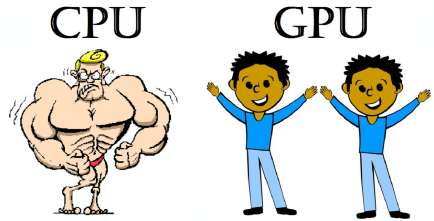


CPU vs. GPU

- The CPU operates at a higher frequency compared to the GPU.
- The GPU, however, excels in parallel processing by executing many threads simultaneously.
- On the GPU, cores are organized into groups known as "Streaming Multiprocessors (SM)." Additionally, Nvidia GPUs incorporate a "Tensor Processing Unit (TPU)" to optimize AI and machine learning computations.
- Threads on the GPU are executed in groups; typically, each group consists of 32 threads, referred to as "warps" in Nvidia GPUs and "wavefronts" in AMD GPUs.

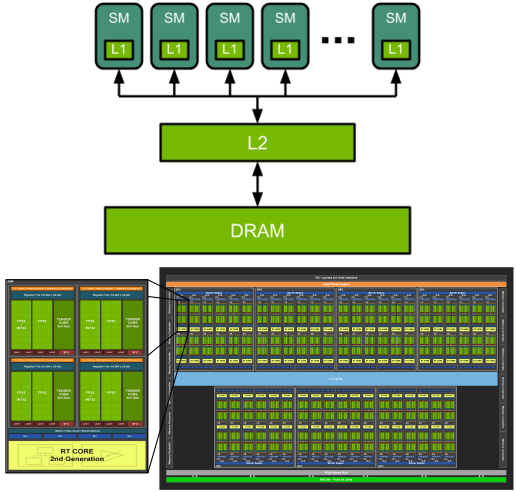


Source:Nvidia: CUDA programming



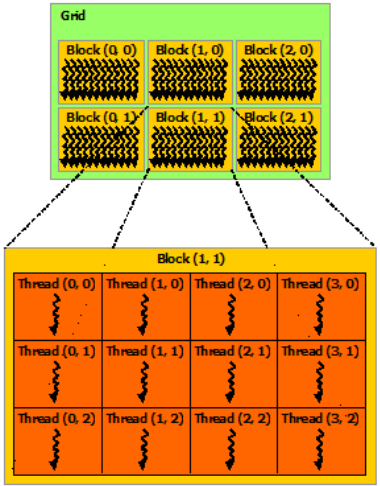
GPU Architecture

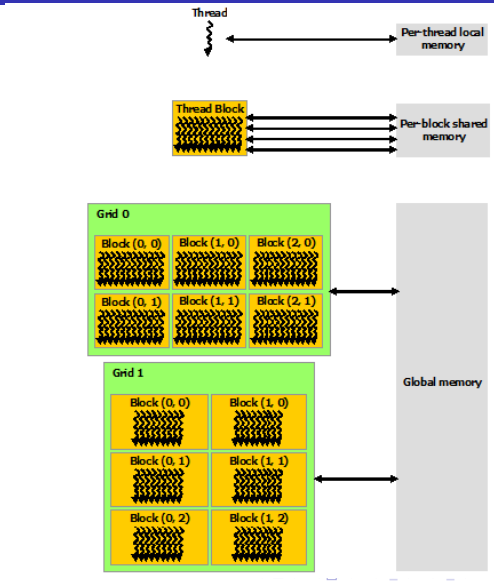
- The Ampere GPU features seven Graphics Processing Clusters (GPCs), 42 Tensor Processing Clusters (TPCs), and 84 Streaming Multiprocessors (SMs).
- The Volta GPU consists of six GPCs, each containing seven TPCs, totaling 14 SMs.
- Each SM is equipped with L1 cache (up to 128 KB), while L2 cache (up to 6144 KB) is shared among the GPCs.
- Dedicated Ray Tracing (RT) cores enhance performance for ray-tracing rendering computations.
- Tensor Cores offer significant speedups for AI neural network training computations.
- Programmable Shading Cores incorporate CUDA cores for parallel processing power.



Source: Nvidia: Deep Learning

Thread hierarchy

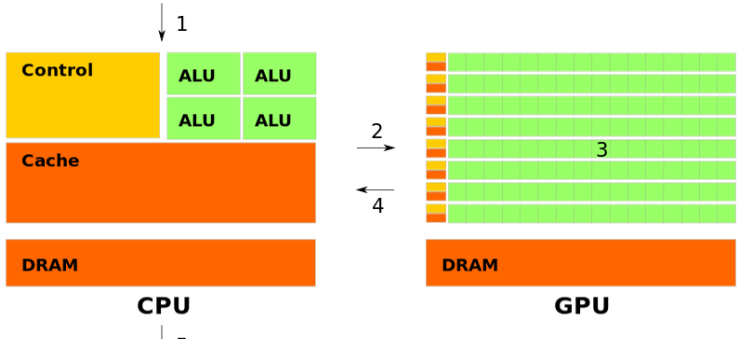




Outline for section 5

How GPUs are Utilized for Computation

- **Step 1: Application Preparation** - Initialize memory spaces on both the CPU and GPU.
- **Step 2: Data Transfer** - Transfer data from the CPU to the GPU.
- **Step 3: Computation** - Perform computations utilizing the parallel processing capabilities of the GPU.
- **Step 4: Data Transfer Back** - Transfer the computed results back to the CPU.
- **Step 5: Cleanup** - Finalize the application and free memory on both the CPU and GPU.



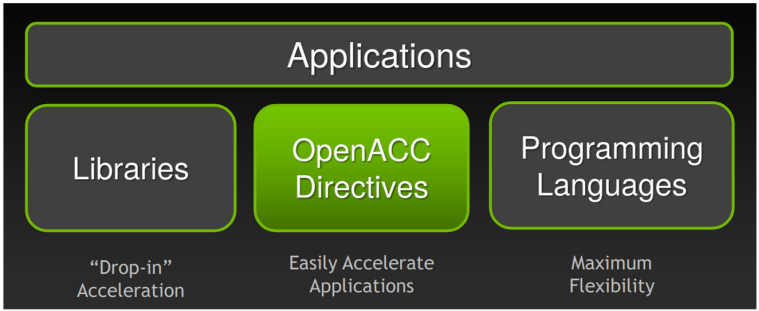
Outline for section 6

Key Points About OpenACC

- OpenACC is a directive-based programming model rather than a dedicated GPU programming language.
- It provides a means to express parallelism in your code.
  - Supports languages such as C, C++, and Fortran.
- OpenACC is compatible with both NVIDIA and AMD GPUs.
  - Allows for single-source code implementation, eliminating the need to alter code when switching between GPU vendors.
- The latest version of the OpenACC API includes comprehensive definitions of directives and clauses for effective programming.

"OpenACC will enable programmers to develop portable applications that maximize easily the performance and power efficiency benefits of the hybrid CPU/GPU architecture of Titan." - *Buddy Bland, Titan Project Director, Oak Ridge National Lab.*

Ways to Accelerate Applications on the GPU



"OpenACC is a technically impressive initiative brought together by members of the OpenMP Working Group on Accelerators, as well as many others. We look forward to releasing a version of this proposal in the next release of OpenMP." - *Michael Wong, CEO of OpenMP Directives Board.*

Accelerating Applications on GPUs

- **Libraries:** User-friendly options that require minimal GPU programming expertise.
  - Examples: cuBLAS, cuFFT, CUDA Math Library, etc.
- **Directive-Based Programming:** Enhances application performance by incorporating directives into existing code.
  - Examples: OpenACC, OpenMP Offloading.
- **Programming Languages:** Low-level languages that allow for deeper optimization of applications on GPU accelerators.
  - Examples: CUDA, OpenCL.

Compilers

- OpenACC is supported by several commercial and open-source compilers, with the following being among the most widely used:
  - PGI
  - GCC
  - HPE Gray (available for FORTRAN only).
- PGI compilers are now part of Nvidia and can be accessed through the Nvidia HPC SDK.
- For today's practical session, we will be utilizing the Nvidia HPC SDK compiler.

Compiler Options

- `-acc`: Enables recognition of OpenACC directives. OpenACC can generate code for multicore CPUs, similar to OpenMP.
  - `-acc=gpu` Build for GPU acceleration.
  - `-acc=multicore` Build for multicore CPU (multithreaded) execution.
  - `-acc=host` Build for single-core CPU (sequential) execution.
  - `-acc=noautopar` Disable automatic parallelization within parallel regions (default is `-acc=autopar`).
- `-gpu`: Specifies GPU-related options for the compiler.
  - `-gpu=ccXX` Specify the compute capability for which the code should be built; for example, `cc80` for compute capability 8.0.
  - `-gpu=managed` Enable NVIDIA Unified Memory, which simplifies data management but may introduce failures in some cases.
  - `-gpu=pinned` Activate pinned memory to enhance the performance of data transfers.
  - `-gpu=lineinfo` Generate debugging line information with less overhead than the `-g` option.

Profiling Information During Compilation

- `-Minfo`: Displays information about the optimizations applied by the compiler.
- `-Minfo=accel`: Provides detailed information specifically about OpenACC. (Mandatory for this training course!)
- `-Minfo=all`: Outputs comprehensive information on all optimizations, including OpenACC, vectorization, and Fused Multiply-Add (FMA). This option is recommended for in-depth analysis.

C Compilation Command

```
nvc -fast -Minfo=accel -acc=gpu -gpu=cc80 Test.c
```

C++ Compilation Command

```
nvc++ -fast -Minfo=accel -acc=gpu -gpu=cc80 Test.cc
```

FORTRAN Compilation Command

```
nvfortran -fast -Minfo=accel -acc=gpu -gpu=cc80 Test.f90
```

Device information

$ nvaccelinfo

```
CUDA Driver Version:           12000
NVRM version:                  NVIDIA UNIX x86_64 Kernel Module  525.85.12
                               Sat Jan 28 02:10:06 UTC 2023

Device Number:                 0
Device Name:                   Tesla V100-SXM2-32GB
Device Revision Number:        7.0
Global Memory Size:            34079637504
Number of Multiprocessors:     80
Concurrent Copy and Execution: Yes
Total Constant Memory:         65536
Total Shared Memory per Block: 49152
Registers per Block:           65536
Warp Size:                     32
Maximum Threads per Block:     1024
Maximum Block Dimensions:      1024, 1024, 64
Maximum Grid Dimensions:       2147483647 x 65535 x 65535
```

OpenACC Directives

- **Compute Constructs:**
  - *parallel*, *kernel* - Define a parallel region to execute computations concurrently.
- **Loop Constructs:**
  - *loop*, *collapse*, *gang*, *worker*, *vector* - Optimize thread utilization for work-sharing within loops.
- **Data Management Clauses:**
  - *copy*, *create*, *copyin*, *copyout*, *delete*, *present* - Manage data transfers between the host and device effectively.
- **Additional Directives:**
  - *reduction*, *atomic*, *cache* - Implement special operations to enhance performance while maintaining parallelism.

For further information about OpenACC directives, please refer to this resource.

Basic Programming Structure

Example in C/C++:

```
// C/C++
#include "openacc.h"
// ACC Directives
#pragma acc <directive> [clauses [[,] clause] . . .]
<code>
// Additional code
```

Example in Fortran:

```
!! Fortran
use openacc
!$acc <directive> [clauses [[,] clause] . . .]
<code>
! Additional code
```

Outline for section 7

Compute Constructs

- `parallel`
  - Similar in structure to OpenMP directives.
  - Control of the parallel region is the programmer's responsibility.
  - Caution is advised for beginners to avoid data races and ensure correct computations.

- `kernels`
  - Compilers optimize parallelization within the `kernels` scope.
  - This allows beginners to leverage parallelizing features with ease.
  - Careful management of data movement is essential to avoid issues such as pointer aliasing, which can prevent effective parallelization (e.g., using `restrict`).

Kernels in C/C++

```c
// Hello_World.c
void Print_Hello_World()
{
  for(int i = 0; i < 5; i++)
  {
    printf("Hello World!\n");
  }
}
```

```c
// Hello_World_OpenACC.c
void Print_Hello_World()
{
#pragma acc kernels loop
  for(int i = 0; i < 5; i++)
  {
    printf("Hello World!\n");
  }
}
```

Compilation

To compile the OpenACC program, use the following command:

```
nvc -fast -Minfo=all -acc=gpu -gpu=cc80 Hello_World_OpenACC.c
```

This command activates optimizations for GPU execution. Review the compiler output for useful information regarding the kernel execution.

Kernels in FORTRAN

```
!! Hello_World.f90
subroutine Print_Hello_World()
  integer :: i

  ! Loop to print "hello world" five times
  do i = 1, 5
     print *, "hello world"
  end do

end subroutine Print_Hello_World
```

```
!! Hello_World_OpenACC.f90
subroutine Print_Hello_World()
  integer :: i
  !$acc kernels loop
  ! Loop to offload computation to GPU
  do i = 1, 5
    print *, "hello world"
  end do
 !$acc end kernels
end subroutine Print_Hello_World
```

Compilation

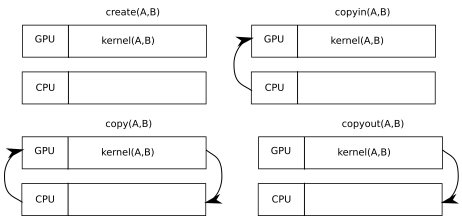
To compile the OpenACC version, use the following command: `nvfortran -fast -Minfo=all -acc=gpu Hello_World_OpenACC.f90`

Here, `-acc` refers to target-specific switches, and `-gpu=cc80` indicates the GPU compute capability.

Outline for section 8

Data Management (Structured)

- `copyin(list)` - Allocates memory on the GPU and transfers data from the CPU (host) to the GPU upon entering a parallel region.
- `copyout(list)` - Allocates memory on the GPU and transfers data back to the CPU (host) upon exiting a parallel region.
- `copy(list)` - Allocates memory on the GPU, copies data from the CPU (host) to the GPU when entering a region, and transfers data back to the CPU (host) when exiting.
- `create(list)` - Allocates memory on the GPU without transferring any data.
- `delete(list)` - Deallocates memory on the GPU without transferring data back to the CPU.
- `present(list)` - Indicates that data is already available on the GPU from another region.



Loop and Data Clauses in Vector Addition in C/C++

Vector Addition in Standard C/C++:

```
1   // Vector_Addition.c
2   float * Vector_Addition
3   (float *restrict a, float *restrict b,
4    float *restrict c, int n)
5   {
6
7
8     for(int i = 0; i < n; i ++)
9       {
10        c[i] = a[i] + b[i];
11      }
12    return c;
13  }
```

Vector Addition with OpenACC:

```
1   // Vector_Addition_OpenACC.c
2   float * Vector_Addition
3   (float *restrict a, float *restrict b,
4   float *restrict c, int n)
5   {
6   #pragma acc kernels loop
7   copyin(a[0:n], b[0:n]) copyout(c[0:n])
8     for(int i = 0; i < n; i ++)
9       {
10        c[i] = a[i] + b[i];
11      }
12
13  }
```

```
$ nvc -fast -Minfo=accel -acc=gpu -gpu=cc80 Vector_Addition_OpenACC.c
Vector_Addition:
     14, Generating copyin(a[:n]) [if not already present]
         Generating copyout(c[:n]) [if not already present]
         Generating copyin(b[:n]) [if not already present]
     16, Loop is parallelizable
         Generating Tesla code
         16, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

Loop and Data Clauses in Vector Addition in FORTRAN

Vector Addition with FORTRAN:

```
1  !! Vector_Addition.f90
2  module Vector_Addition_Mod
3    implicit none
4  contains
5    subroutine Vector_Addition(a, b, c, n)
6      !! Input vectors
7      real(8), intent(in), dimension(:) :: a,b
8      real(8), intent(out), dimension(:) :: c
9      integer :: i, n
10
11
12     do i = 1, n
13        c(i) = a(i) + b(i)
14     end do
15
16   end subroutine Vector_Addition
17 end module Vector_Addition_Mod
```

Vector Addition in Standard FORTRAN:

```
1  | !! Vector_Addition_OpenACC.f90
2  | module Vector_Addition_Mod
3  |   implicit none
4  | contains
5  |   subroutine Vector_Addition(a, b, c, n)
6  |     !! Input vectors
7  |     real(8), intent(in), dimension(:) :: a,b
8  |     real(8), intent(out), dimension(:) :: c
9  |     integer :: i, n
10 |     !$acc kernels loop copyin(a(1:n), b(1:n))
11 |     copyout(c(1:n))
12 |     do i = 1, n
13 |        c(i) = a(i) + b(i)
14 |     end do
15 |     !$acc end kernels
16 |   end subroutine Vector_Addition
17 | end module Vector_Addition_Mod
```

```
$ nvfortran -fast -Minfo=accel -acc=gpu -gpu=cc80 Vector_Addition_OpenACC.f90
vector_addition:
     12, Generating copyin(a(:n)) [if not already present]
         Generating copyout(c(:n)) [if not already present]
         Generating copyin(b(:n)) [if not already present]
     13, Loop is parallelizable
         Generating Tesla code
         13, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
```

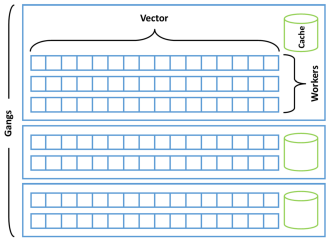
Data Transfer Using High-Level API (Structured Data Mapping)

- **Ease of Use:** The mapping approach is user-friendly, using `#pragma acc parallel [clause ...] new-line` with a clause like `copyin(data)`.
- **Scope Limitations:** Data is only valid within the OpenACC compute constructs' scope.
- **Computational Constraints:** This approach may not benefit extensive computations, as data is deallocated or returned to the host after the computation ends.

Outline for section 9

Loop Optimization

CUDA	OpenACC (2.7 API)
Thread Blocks	`num_gangs()`
Warps	`num_workers()`
Threads	`vector_length()`



- OpenACC facilitates mapping of threads to the SIMT architecture.
- Programmers have the flexibility to define the number of threads using OpenACC APIs.
- The compiler optimizes thread selection based on architecture and problem requirements.
- Manual optimization of thread settings is recommended for enhanced performance.

Example - Loop Optimization

Example in FORTRAN:

```
!$acc data copyin(a(1:n), b(1:n)) copyout(c(1:n))
!$acc parallel loop num_gangs(128) vector_length(128)
    do i = 1, n
       c(i) = a(i) + b(i)
    end do
!$acc end parallel
!$acc end data
```

Example in C/C++:

```
#pragma acc data copyin(a[0:n], b[0:n]) copyout(c[0:n])
  {
#pragma acc parallel loop num_gangs(128) vector_length(128)
    for(int i = 0; i < n; i ++)
      {
        c[i] = a[i] + b[i];
      }
  }
```

Example - Loop Optimization

Example in FORTRAN: Compilation Output

```
nvfortran -fast -Minfo=accel -acc=gpu -gpu=cc80 Vector_Addition_OpenACC_Optimized.f90
vector_addition:
     12, Generating copyin(a(:n)) [if not already present]
         Generating copyout(c(:n)) [if not already present]
         Generating copyin(b(:n)) [if not already present]
     13, Generating Tesla code
         14, !$acc loop gang(128), vector(128) ! blockidx%x threadidx%x
```

Example in C/C++: Compilation Output

```
nvc -fast -Minfo=accel -acc=gpu -gpu=cc80 Vector_Addition_OpenACC_Optimized.c
Vector_Addition:
     16, Generating copyin(a[:n]) [if not already present]
         Generating copyout(c[:n]) [if not already present]
         Generating copyin(b[:n]) [if not already present]
         Generating Tesla code
         18, #pragma acc loop gang(128), vector(128) /* blockIdx.x threadIdx.x */
```

Example - Loop Optimization

- Both examples demonstrate data mapping using High-Level API using OpenACC.
- Ensure proper data transfers using ‘copyin‘ and ‘copyout‘.
- Parallelization is managed by setting the number of gangs and vector length, optimizing performance in parallel computing environments.

Outline for section 10

Collapse Clause

- Facilitates efficient parallelization in nested loops.
- Distributes the total number of iterations across the available threads.
- Example: When the outer loop index equals the number of threads, the workload of the innermost loop is divided among the threads, enhancing performance.
- Only one collapse clause is permitted within a given nested loop structure.

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

Collapse Clause in C/C++

```c
// Matrix_Multiplication.c
for(int row = 0; row < width ; ++row)
    {
      for(int col = 0; col < width ; ++col)
        {
          sum=0;
          for(int i = 0; i < width ; ++i)
            {
              sum += a[row*width+i]
                   * b[i*width+col];
            }
          c[row*width+col] = sum;
        }
    }
```

```c
// Matrix_Multiplication_OpenACC.c
pragma acc kernels loop collapse(2)
                        reduction(+:sum)
  for(int row = 0; row < width ; ++row)
     {
       for(int col = 0; col < width ; ++col)
         {
           sum=0;
           for(int i = 0; i < width ; ++i)
             {
               sum += a[row*width+i]
                    * b[i*width+col];
             }
           c[row*width+col] = sum;
         }
     }
```

```
nvc -fast -Minfo=accel -acc=gpu -gpu=cc80 Matrix_Multiplication_OpenACC.c
Matrix_Multiplication:
      9, Generating copyin(a[:width*width],b[:width*width]) [if not already present]
         Generating copyout(c[:width*width]) [if not already present]
         Generating create(sum) [if not already present]
     11, Loop carried dependence of c-> prevents parallelization
         Loop carried backward dependence of c-> prevents vectorization
     13, Loop is parallelizable
         Generating Tesla code
         11, #pragma acc loop seq collapse(2)
         13,   collapsed */
             Generating reduction(+:sum)
         16, #pragma acc loop vector(128) /* threadIdx.x */
             Generating implicit reduction(+:sum)
```

Collapse Clause in FORTRAN

```
1    !! Matrix_Multiplication.f90
2        do row = 0, width-1
3           do col = 0, width-1
4              sum=0
5              do i = 0, width-1
6                 sum = sum + (a((row*width)+i+1)
7                           * b((i*width)+col+1))
8              enddo
9              c(row*width+col+1) = sum
10          enddo
11       enddo
```

```
1    !! Matrix_Multiplication_OpenACC.f90
2       !$acc loop collapse(2) reduction(+:sum)
3       do row = 0, width-1
4          do col = 0, width-1
5             sum=0
6             do i = 0, width-1
7                sum = sum + (a((row*width)+i+1)
8                          * b((i*width)+col+1))
9             enddo
10            c(row*width+col+1) = sum
11         enddo
12      enddo
13      !$acc end loop
```

```
nvfortran -fast -Minfo=accel -acc=gpu -gpu=cc80 Matrix_Multiplication_OpenACC.f90
matrix_multiplication:
     14, Generating copyin(a(:width*width),b(:width*width)) [if not already present]
         Generating copyout(c(:width*width)) [if not already present]
         Generating create(sum) [if not already present]
         Generating Tesla code
         16, !$acc loop gang collapse(2) ! blockidx%x
             Generating reduction(+:sum)
         17,   ! blockidx%x collapsed
         19, !$acc loop vector(128) ! threadidx%x
             Generating implicit reduction(+:sum)
             19, Loop is parallelizable
```

Reduction Clause

- Reduction is essential in OpenACC for performing operations like summing or incrementing values into a shared scalar variable across parallel iterations.
- Syntax: `reduction(operators: variable)`
  - Supports various arithmetic reductions: `+`, `*`, `max`, `min`, etc.
- Particularly advantageous in applications like matrix-vector and matrix-matrix multiplication.

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

Reduction Clause in C/C++

```c
// Matrix_Multiplication.c
for(int row = 0; row < width ; ++row)
    {
      for(int col = 0; col < width ; ++col)
        {
          sum=0;
          for(int i = 0; i < width ; ++i)
            {
              sum += a[row*width+i]
                   * b[i*width+col];
            }
          c[row*width+col] = sum;
        }
    }
```

```c
// Matrix_Multiplication_OpenACC.c
pragma acc kernels loop collapse(2)
                        reduction(+:sum)
  for(int row = 0; row < width ; ++row)
     {
       for(int col = 0; col < width ; ++col)
         {
           sum=0;
           for(int i = 0; i < width ; ++i)
             {
               sum += a[row*width+i]
                    * b[i*width+col];
             }
           c[row*width+col] = sum;
         }
     }
```

```
nvc -fast -Minfo=accel -acc=gpu -gpu=cc80 Matrix_Multiplication_OpenACC.c
Matrix_Multiplication:
      9, Generating copyin(a[:width*width],b[:width*width]) [if not already present]
         Generating copyout(c[:width*width]) [if not already present]
         Generating create(sum) [if not already present]
     11, Loop carried dependence of c-> prevents parallelization
         Loop carried backward dependence of c-> prevents vectorization
     13, Loop is parallelizable
         Generating Tesla code
         11, #pragma acc loop seq collapse(2)
         13,   collapsed */
             Generating reduction(+:sum)
         16, #pragma acc loop vector(128) /* threadIdx.x */
             Generating implicit reduction(+:sum)
```

Reduction Clause in FORTRAN

```
!! Matrix_Multiplication.f90
   do row = 0, width-1
      do col = 0, width-1
         sum=0
         do i = 0, width-1
            sum = sum + (a((row*width)+i+1)
                      * b((i*width)+col+1))
         enddo
         c(row*width+col+1) = sum
      enddo
   enddo
```

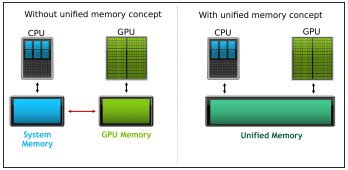
```
!! Matrix_Multiplication_OpenACC.f90
   !$acc loop collapse(2) reduction(+:sum)
   do row = 0, width-1
      do col = 0, width-1
         sum=0
         do i = 0, width-1
            sum = sum + (a((row*width)+i+1)
                      * b((i*width)+col+1))
         enddo
         c(row*width+col+1) = sum
      enddo
   enddo
   !$acc end loop
```

```
nvfortran -fast -Minfo=accel -acc=gpu -gpu=cc80 Matrix_Multiplication_OpenACC.f90
matrix_multiplication:
     14, Generating copyin(a(:width*width),b(:width*width)) [if not already present]
         Generating copyout(c(:width*width)) [if not already present]
         Generating create(sum) [if not already present]
         Generating Tesla code
         16, !$acc loop gang collapse(2) ! blockidx%x
             Generating reduction(+:sum)
         17,   ! blockidx%x collapsed
         19, !$acc loop vector(128) ! threadidx%x
             Generating implicit reduction(+:sum)
     19, Loop is parallelizable
```

Outline for section 11

Unified Memory in OpenACC

- Simplifies data management between CPU and GPU.
- No need for manual data transfers — OpenACC handles it automatically.
- Enabled with the compiler flag `-gpu=managed`.



Enabling Unified Memory

C/C++

```
nvc -fast -acc=gpu -gpu=cc80 -gpu=managed -Minfo=accel test.c
```

Fortran

```
nvfortran -fast -acc=gpu -gpu=cc80 -gpu=managed -Minfo=accel test.f90
```

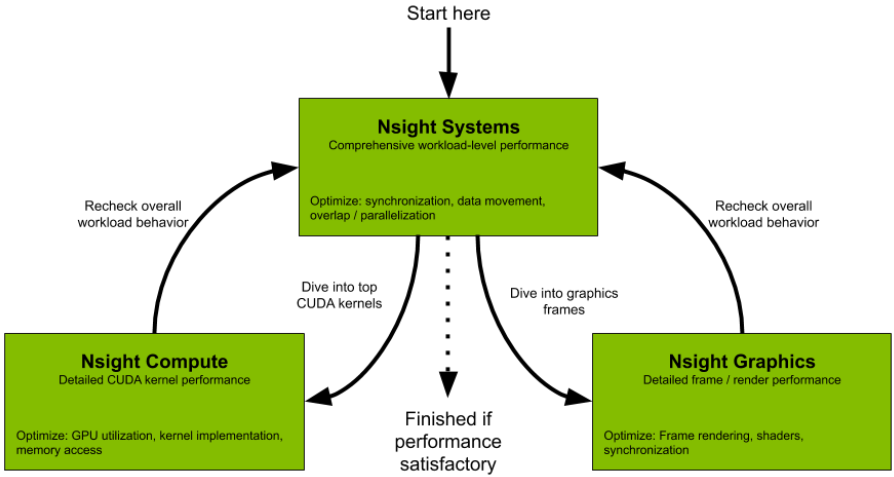
Unified Memory in OpenACC

Without Unified Memory	**With Unified Memory**
Allocate the host memory	Allocate the host memory
Initialize the host value	Initialize the host value
Use data clauses, e.g., `copy`, `copyin`	~~Use data clauses, e.g., `copy`, `copyin`~~
Do the computation using the GPU kernel	Do the computation using the GPU kernel
Free host memory	Free host memory

- It simplifies programming for developers by allowing them to focus on the implementation of algorithms rather than data management.
- However, it is important to note that data must still be transferred between the CPU and GPU whenever it is required for computation.
- **Limitations**: This approach is currently applicable only when utilizing a single GPU.

Outline for section 12

Profiling


Start here
Nsight Systems
Comprehensive workload-level performance
Optimize: synchronization, data movement, overlap / parallelization
Recheck overall workload behavior
Recheck overall workload behavior
Dive into top CUDA kernels
Dive into graphics frames
Nsight Compute
Detailed CUDA kernel performance
Optimize: GPU utilization, kernel implementation, memory access
Finished if performance satisfactory
Nsight Graphics
Detailed frame / render performance
Optimize: Frame rendering, shaders, synchronization

Profiling (without GUI)

- `export NVCOMPILER_ACC_TIME=1`
  - if you plan to use another profiling tool, please set `NVCOMPILER_ACC_TIME=0`
- `export NVCOMPILER_ACC_NOTIFY=3`
  - 1: kernel launches
  - 2: data transfers
  - 4: region entry/exit
  - 8: wait for operations or synchronizations
  - 16: device memory allocates and deallocates

Keeping `NVCOMPILER_ACC_NOTIFY=3` provides kernel executions and data transfer information.

```
/mnt/irisgpfs/users/ekrishnasamy/ULHPC-School/Vector_Addition_OpenACC.f90
  vector_addition  NVIDIA  devicenum=0
    time(us): 66
    12: compute region reached 1 time
        12: kernel launched 1 time
            grid: [128]  block: [128]
            elapsed time(us): total=45 max=45 min=45 avg=45
    12: data region reached 2 times
        12: data copyin transfers: 2
             device time(us): total=38 max=23 min=15 avg=19
        16: data copyout transfers: 1
             device time(us): total=28 max=28 min=28 avg=28
```

Profiling (with GUI)

```
[u100@mel2073 Vector-addition]$ nvc -fast -acc=gpu -gpu=cc80 -Minfo=accel Vector-addition.c
nvc-Warning-CUDA_HOME has been deprecated. Please use NVHPC_CUDA_HOME instead.
[u100@mel2073 Vector-addition]$ nsys profile -o timeline ./a.out
Warning: LBR backtrace method is not supported on this platform. DWARF backtrace method will be used.
This program does the addition of two vectors
Please specify the vector size = 10000
PASSED
Generating '/tmp/nsys-report-6c02.qdstrm'
[1/1] [========================100%] timeline.nsys-rep
```

```
// Open the GUI application and load the timeline.nsys-rep
$ nsys-ui &
```