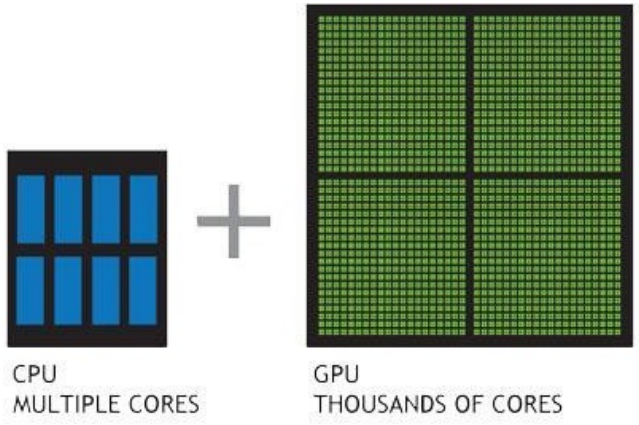
Parallel Programming with OpenACC

U NIT-IV

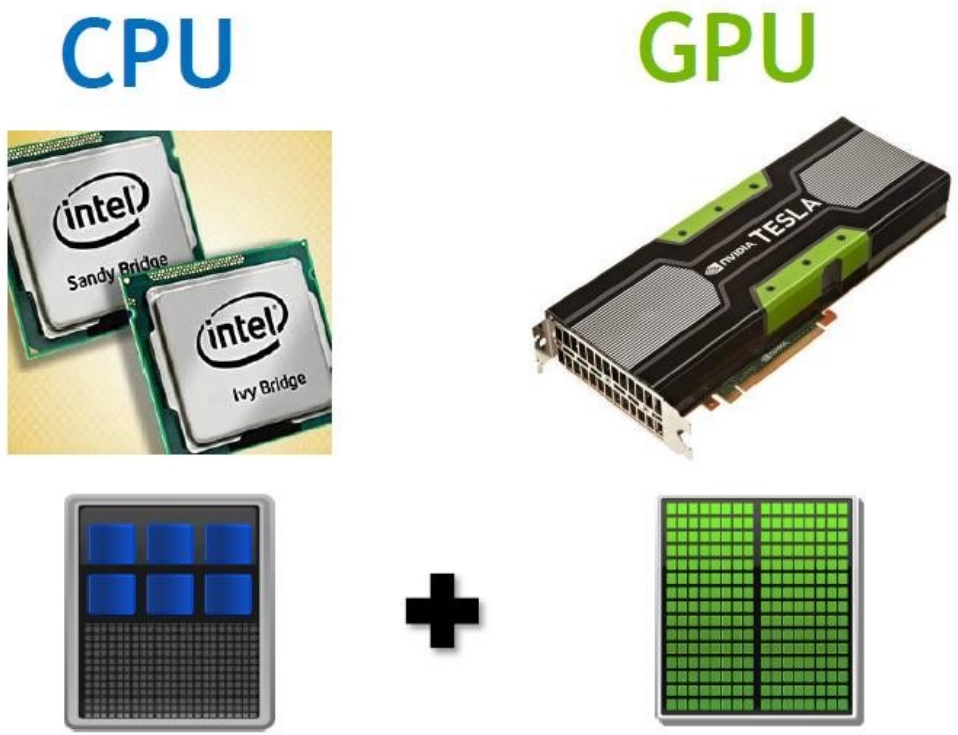
GPU and GPGPU

- Originally, graphics processing unit (GPU) is dedicated for manipulating computer graphics and image processing. Traditionally GPU is known as “video card”.
- GPU’s highly parallel structure makes it efficient for parallel programs. Nowadays GPUs are used for tasks that were formerly the domain of CPUs, such as scientific computation. This kind of GPU is called general-purpose GPU (GPGPU) .
- In many cases, a parallel program runs faster on GPU than on CPU. Note that a serial program runs faster on CPU than on GPU.
- The most popular type of GPU in the high-performance computing world is NVIDIA GPU. We will only focus on NVIDIA GPU here.

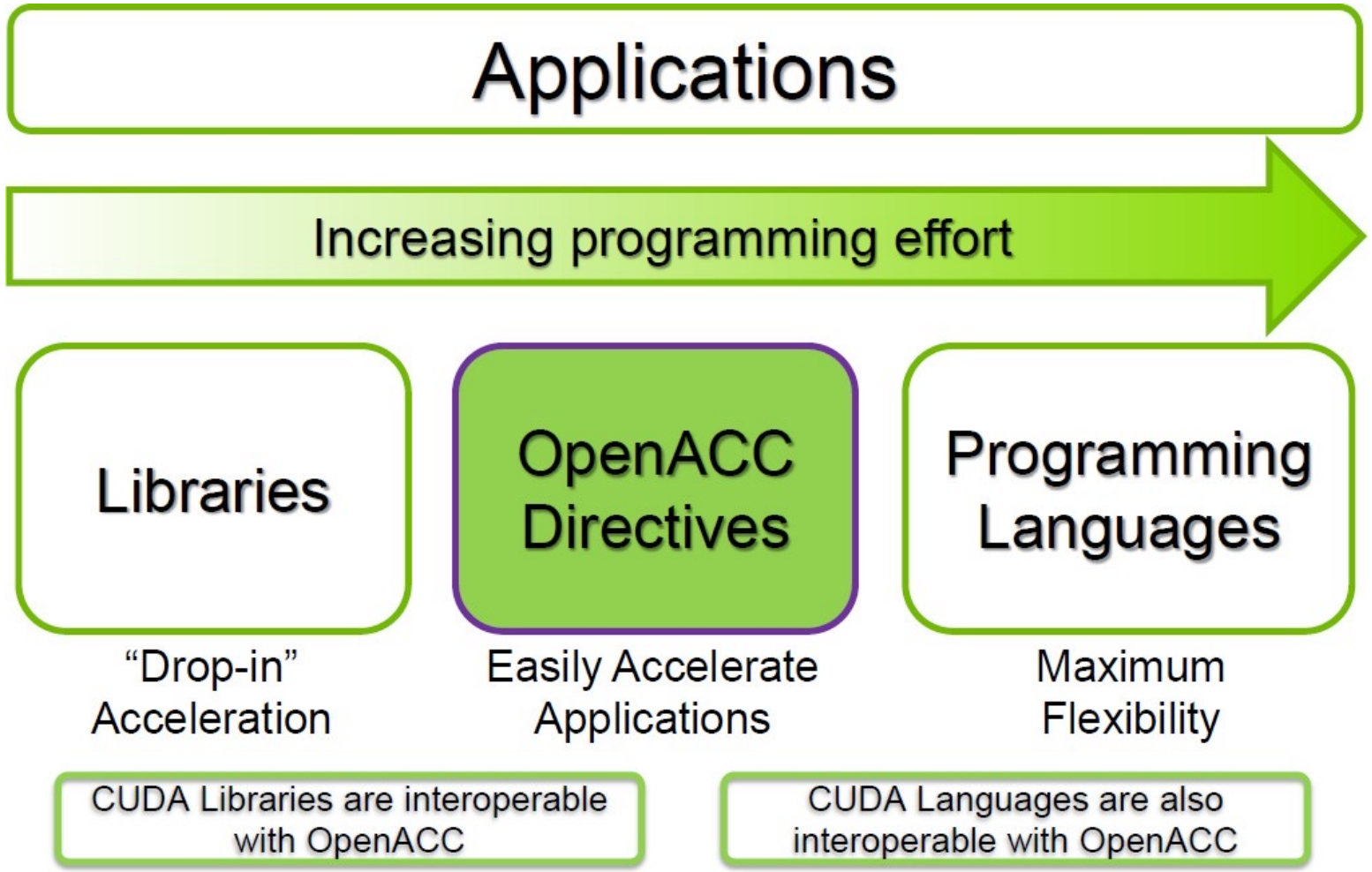


GPU is an accelerator

- GPU is a device on a CPU-based system. GPU is connected to CPU through PCI bus.
- Computer program can be parallelized and accelerated on GPU.
- CPU and GPU has separated memories. Data transfer between CPU and GPU is required for programming.



Three ways to accelerate applications on GPU



- The OpenACC Application Programming Interface (API) provides a set of compiler directives, library routines, and environment variables that can be used to write data-parallel FORTRAN, C, and C++ programs that run on accelerator devices, including GPUs.
- It is an extension to the host language.
- The OpenACC specification was initially developed by the Portland Group (PGI), Cray Inc., and NVIDIA, with support from CAPS enterprise.

What is OpenACC

- OpenACC (for Open Accelerators) is a programming standard for parallel computing on accelerators (mostly on NIVDIA GPU).
- It is designed to simplify GPU programming.
- The basic approach is to insert special comments (directives) into the code so as to offload computation onto GPUs and parallelize the code at the level of GPU (CUDA) cores.
- It is possible for programmers to create an efficient parallel OpenACC code with only minor modifications to a serial CPU code.

What are compiler directives?

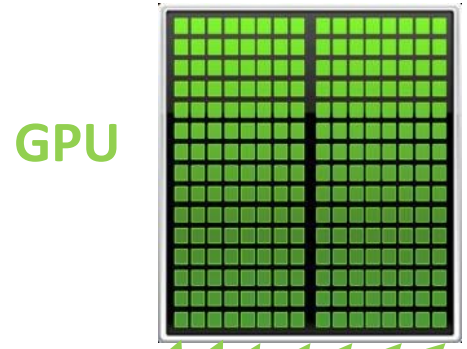
❑ The directives tell the compiler or runtime to ……

✓ Generate parallel code for GPU

✓ Allocate GPU memory and copy input data

✓ Execute parallel code on GPU

✓ Copy output data to CPU and deallocate GPU memory



❑ The first OpenACC directive: kernels

✓ ask the compiler to generate a GPU code

✓ let the compiler determine safe parallelism and data transfer .

```
// … serial code …
#pragma acc kernels
for (int i= 0; i<n; i++) {
   //… parallel code …
}
// … serial code …
```

OpenACC Directive syntax

- C

  #pragma acc directive [clause [,] clause] …]…

  often followed by a structured code block

The first OpenACC program: SAXPY

- Example: Compute *a*x + y*, where *x* and y are vectors, and *a* is a scalar.

C

```
int main(int argc, char **argv){
    int N=1000;
    float a = 3.0f;
    float x[N], y[N];
    for (int i = 0; i < N; ++i) {
        x[i] = 2.0f;
        y[i] = 1.0f;
    }
    #pragma acc  kernels
    for (int i = 0; i < N; ++i) {
        y[i] = a * x[i] + y[i];
    }
}
```

Data dependency

- The loop is not parallelized if there is data dependency. For example,

```
#pragma acc kernels
 for (int i = 0; i < N-1; i++) {
    x[i] = a * x[i+1] ;
 }
```

- The compiling output:

```
……
14, Loop carried dependence of x-> prevents parallelization
      Loop carried backward dependence of x-> prevents vectorization
      Accelerator scalar kernel generated
      Loop carried backward dependence of x-> prevents vectorization
```

- The compiler creates a serial program, which runs slower on GPU than on CPU!

OPENACC VERSUS CUDA C

- One big difference between OpenACC and CUDA C is the use of compiler directives in OpenACC

```
#include<stdio.h>
int main()
{
#pragma acc kernels
for(int i = 0; i < 5; i++)
{
printf("Hello World!\n");
}
}
```

```
1  void computeAcc(float *P, const float *M, const float *N, int Mh,
     int Mw, int Nw)
2  {
3
4   #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw])
      copyout(P[0:Mh*Nw])
5   for (int i=0; i<Mh; i++) {
6       #pragma acc loop
7       for (int j=0; j<Nw; j++) {
8           float sum = 0;
9           for (int k=0; k<Mw; k++) {
10              float a = M[i*Mw+k];
11              float b = N[k*Nw+j];
12              sum += a*b;
13          }
14          P[i*Nw+j] = sum;
15      }
16   }
17 }
```

FIGURE 15.1

Our first OpenACC program.

- Accelerator kernel is generated. The loop computation is offloaded to (Tesla) GPU
  and is parallelized.
- The keywords copy and copyin are involved with data transfer.

- The *loop* will parallelize the *for* loop plus also accommodate other OpenACC *clauses*, for example here *copyin* and *copyout*.
- The above example needs two vectors to be copied to GPU and one vector needs to send the value back to CPU.
- *copyin* will create the memory on the GPU and transfer the data from CPU to GPU.
- *copyout* will create the memory on the GPU and transfer the data from GPU to CPU.

OpenACC brings quite a few benefits to programmers:

- OpenACC programmers can often start with writing a sequential version and then annotate their sequential program with OpenACC directives.
- They can leave most of the heavy lifting to the OpenACC compiler. The details of data transfer between host and accelerator memories, data caching, kernel launching, thread scheduling, and parallelism mapping are all handled by OpenACC compiler and runtime.
- The entry barrier of heterogeneous programmers for accelerators becomes much lower with OpenACC.

- OpenACC provides an incremental path for moving legacy applications to accelerators.
- This is attractive because adding directives disturbs the existing code less than other approaches.
- Some existing scientific applications are large and their developers don’t want to rewrite them for accelerators.
- OpenACC lets these developers keep their applications looking like normal C, C11, or FORTRAN code, and they can go in and put the directives in the code where they are needed one place a time.

- A non-OpenACC compiler is not required to understand and process OpenACC directives, therefore it can just ignore the directives and compile the rest of the program as usual.
- By using the compiler directive approach, OpenACC allows a programmer to write OpenACC programs in such a way that when the directives are ignored, the program can still run sequentially and gives the same result as when the program is run in parallel.
- Parallel programs that have equivalent sequential versions are much easier to debug than those that don't have.
- The matrix multiplication code in Figure 15.1 has this property— the code gives the same result regardless of whether lines 4 and 6 are honored or not.
- Such programs essentially have both the parallel version and the sequential version in one. OpenACC permits a common code base for accelerated and nonaccelerated enabled systems.

- OpenACC is not GPU programming
- OpenACC is expressing the parallelism in your existing code
- OpenACC can be used in both Nvidia and AMD GPUs
- “OpenACC will enable programmers to easily develop portable applications that maximize the performance and power efficiency benefits of the hybrid CPU/GPU architecture of Titan.”
- “OpenACC is a technically impressive initiative brought together by members of the OpenMP Working Group on Accelerators, as well as many others.

OpenACC users need to be aware of the following issues:

- Some OpenACC directives are hints to the OpenACC compiler, which may or may not be able to take full advantage of such hints. Therefore, the performance of an OpenACC program depends more on the capability of the OpenACC compiler used.
- On the other hand, a CUDA C/C11/FORTRAN program expresses parallelism explicitly and relies less on the compiler for parallel performance.
- While it is possible to write OpenACC programs that give the same execution result as when the directives are ignored, this property does not hold automatically for all OpenACC programs. If compiler directives are ignored, some OpenACC programs may give different results or some may not work correctly.

Compilers and directives

- OpenACC is supported by the Nvidia, PGI, GCC, and HPE Gray (only for FORTRAN) compilers
  - Now PGI is part of Nvidia, and it is available through Nvidia HPC SDK
- Compute constructs:
  - parallel and kernel
- Loop constructs:
  - loop, collapse, gang, worker, vector, etc.
- Data management clauses:
  - copy, create, copyin, copyout, delete and present
- Others:
  - reduction, atomic, cache, etc.

The OpenACC Accelerator Model

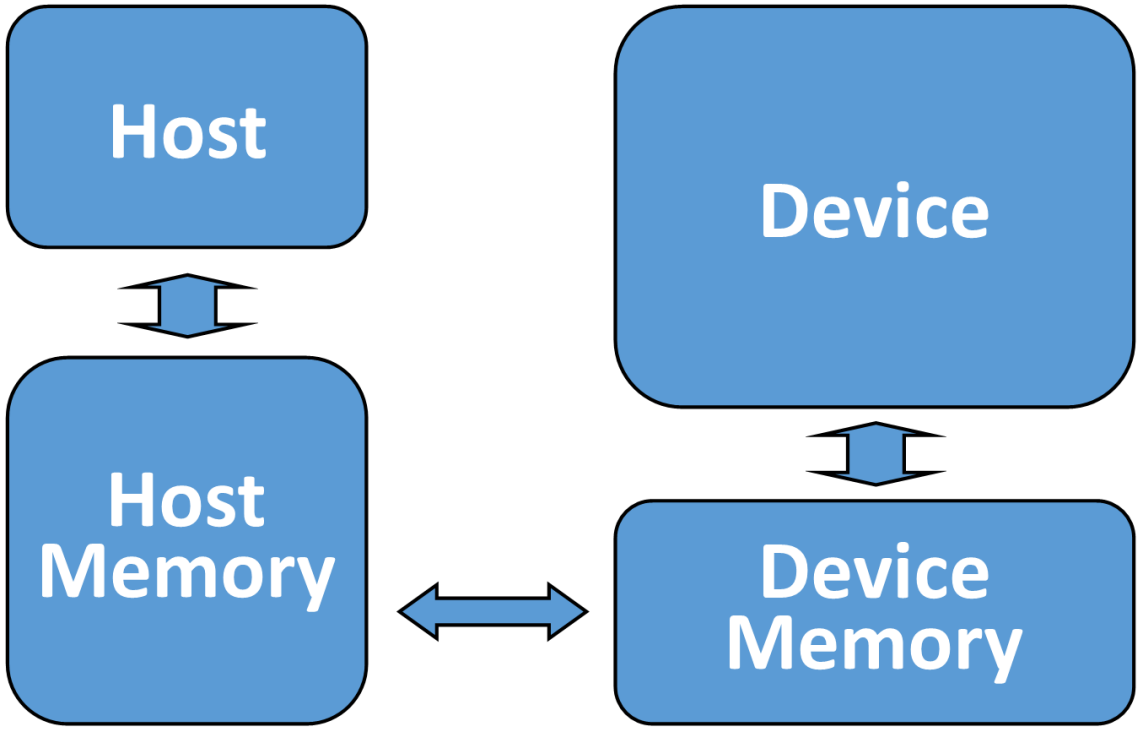


Figure 1.1: OpenACC's Abstract Accelerator Model

- OpenACC would be portable to all computing architectures
- At its core OpenACC supports offloading of both computation and data from a host device to an accelerator device.
- In fact, these devices may be the same or may be completely different architectures, such as the case of a CPU host and GPU accelerator.
- The two devices may also have separate memory spaces or a single memory space.
- In the case that the two devices have different memories the OpenACC compiler and runtime will analyze the code and handle any accelerator memory management and the transfer of data between host and device memory.

EXECUTION MODEL

- The OpenACC target machine has a host and an attached accelerator device, such as a GPU.
- Most accelerator devices can support multiple levels of parallelism.
- Figure 15.2 illustrates a typical accelerator that supports three levels of parallelism.
- At the outermost coarse-grain level, there are multiple execution units. Within each execution unit, there are multiple threads.
- At the innermost level, each thread is capable of executing vector operations.
- Currently, OpenACC does not assume any synchronization capability on the accelerator, except for thread forking and joining.
- Once work is distributed among the execution units, they will execute in parallel from start to finish.
- Similarly, once work is distributed among the threads within an execution unit, the threads execute in parallel.

- An OpenACC program starts its execution on the host single-threaded (Figure 15.3).
- When the host thread encounters a parallel or a kernels construct, a parallel region or a kernels region that comprises all the code enclosed in the construct is created and launched on the accelerator device.
- The parallel region or kernels region can optionally execute asynchronously with the host thread and join with the host thread at a future synchronization point.
- The parallel region is executed entirely on the accelerator device.

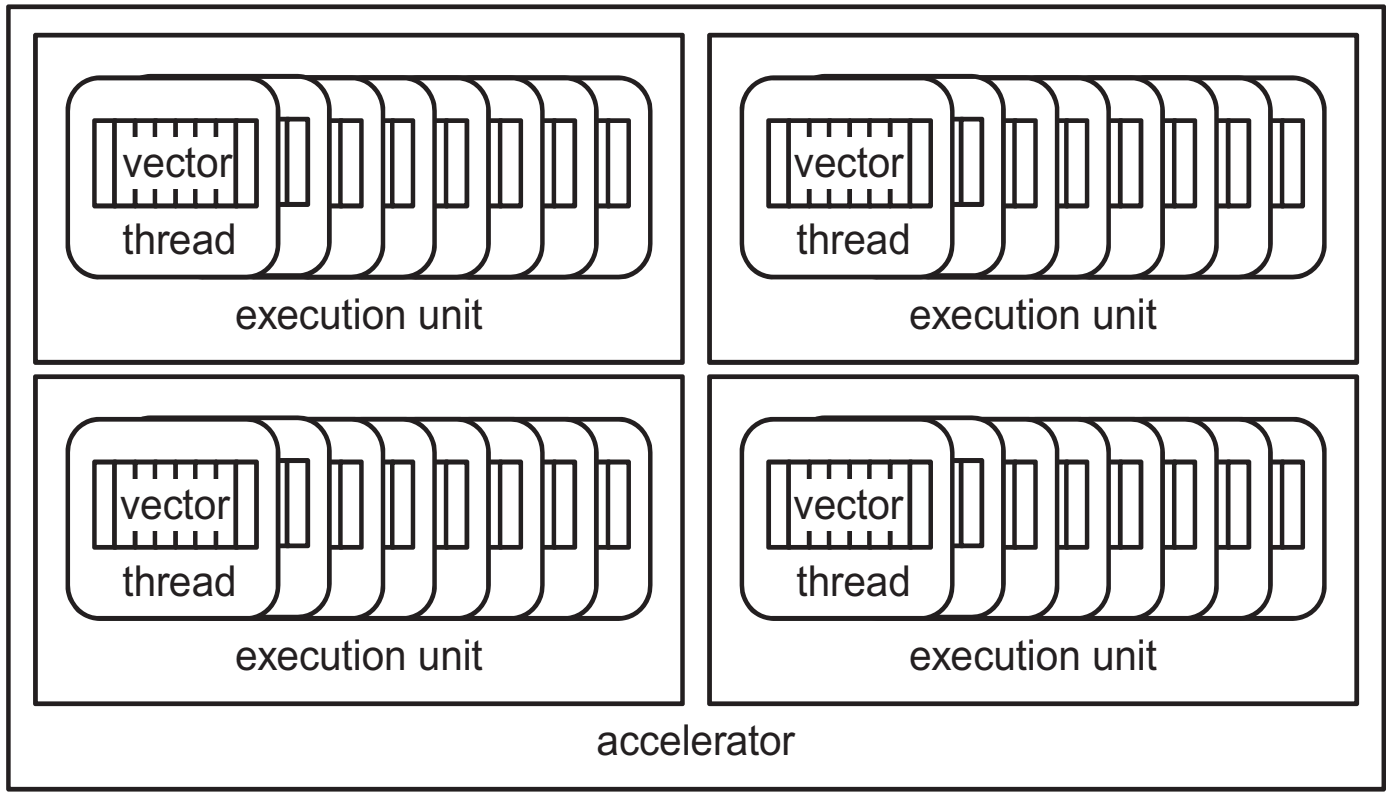


FIGURE 15.2

A typical accelerator device

- The kernels region may contain a sequence of kernels, each of which is executed on the accelerator device.
- The kernel execution follows a fork-join model. A group of gangs are used to execute each kernel.
- A group of workers can be forked to execute a parallel work-sharing loop that belongs to a gang.
- The workers are disbanded when the loop is done.
- Typically a gang executes on one execution unit, and a worker runs on one thread within an execution unit.
- The programmer can instruct how the work within a parallel region or a kernels region is to be distributed among the different levels of parallel- ism on the accelerator.

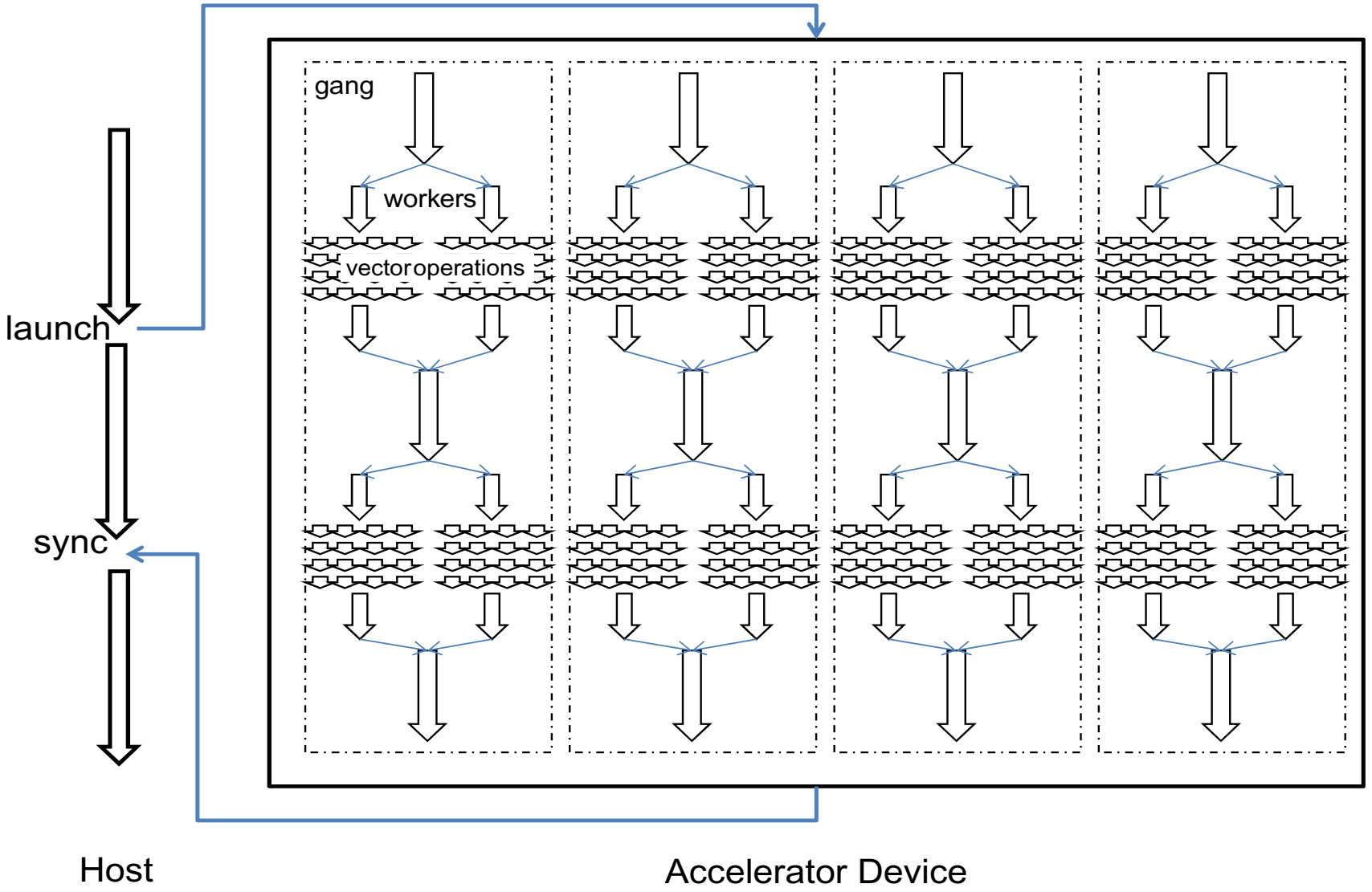


FIGURE 15.3

OpenACC execution model.

MEMORY MODEL

- In an OpenACC memory model, the host memory and the device memory are treated as separated.
- It is assumed that the host is not able to access device memory directly and the device is not able to access host memory directly. This is to ensure that the OpenACC programming model can sup- port a wide range of accelerator devices, including most of the current GPUs that do not have the capability of unified memory access between GPUs and CPUs.
- The unified virtual addressing and the GPUDirect introduced by NVIDIA in CUDA 4.0 allow a single virtual address space for both host memory and device memory and allow direct cross-device mem- ory access between different GPUs. However, cross-host and device memory access is still not possible.

- Just like in CUDA C/C++, in OpenACC input data needs to be transferred from the host to the device before kernel launches and result data needs to be transferred back from the device to the host.
- However, unlike in CUDA C/ C++ where programmers need to explicitly code data movement through API calls, in OpenACC they can just annotate which memory objects need to be transferred, as shown by line 4 in Figure 15.1.
- The OpenACC compiler will automatically generate code for memory allocation, copying, and de-allocation.

- OpenACC adopts a fairly weak consistency model for memory on the accelerator device.
- Although data on the accelerator can be shared by all execution units, OpenACC does not provide a reliable way to allow one execution unit to consume the data produced by another execution unit.
- There are two reasons for this.
- First, recall OpenACC does not provide any mechanism for synchronization between execution units.
- Second, memories between different execution units are not coherent.
- Although some hardware provides instructions to explicitly invalidate and update cache, they are not exposed at the OpenACC level.
- Therefore, in OpenACC, different execution units are expected to work on disjoint memory sets.
- Threads within an execution unit can also share memory and threads have coherent memory.
- However, OpenACC currently only mandates a memory fence at the thread fork and join, which are also the only synchronizations OpenACC provides for threads.
- While the device memory model may appear very limiting, it is not so in practice. For datarace free OpenACC data-parallel applications, the weak memory model works quite well.