OpenACC

Lecture Objectives:

1) Define the concept of gangs and workers as it is related to openAcc.
2) Draw an architecture of a system which uses OpenACC.

OpenACC

- An API consisting of compiler directives for Accelerators.

- Specifies loops and parallel regions that can be sent to attached accelerators (notably GPUs)

- Initially developed by Portland Group (PGI) and NVIDIA with support from CAPS enterprise (http://www.caps-entreprise.com/)

- Similar approach to OpenMP and presently work on merging OpenMP and OpenACC

Sources

These notes introduce OpenACC and are derived from:

Chapter 15 of Programming Massively Parallel Processors by D. B. Kirk and W-M W. Hwu, Morgan Kaufmann, 2013

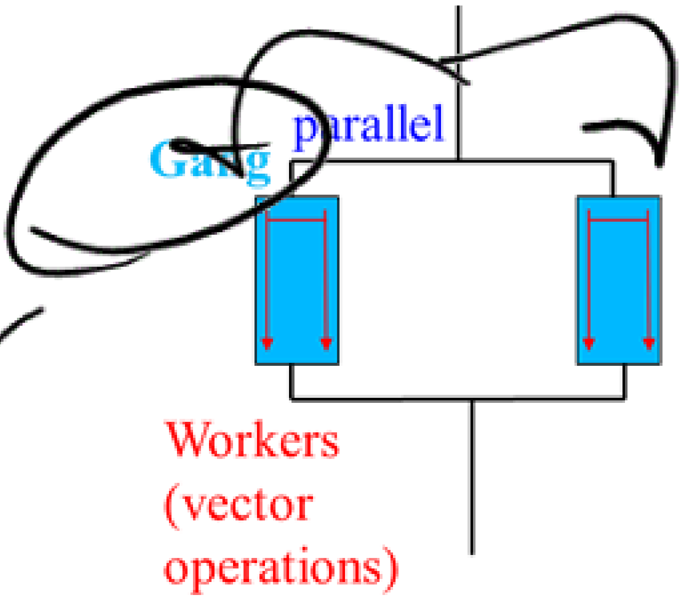
and New

"The OpenACC™ Application Programming Interface," *Version 1.0,* Nov. 2011

http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf

Parallel construct

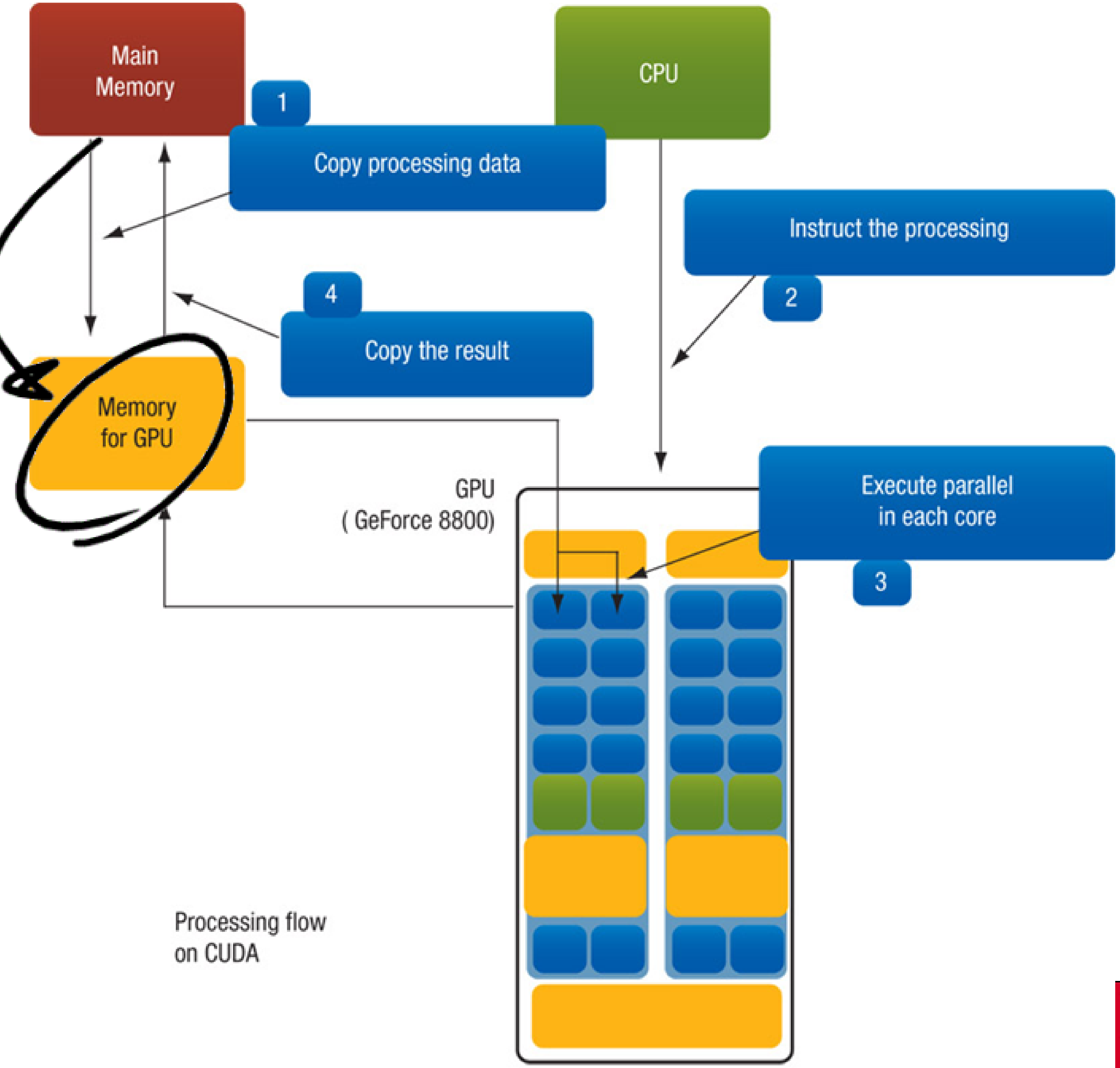
- When program encounters an accelerator **parallel** construct, gangs of workers created to execute accelerator parallel region.
- Once gangs created, number of gangs and number of workers in each gang remain constant for duration of that parallel region.
- One worker in each gang begins executing the code in the structured block of the construct."
- host program will wait until all gangs have completed execution.
"



Gang of workers

CUDA Architecture

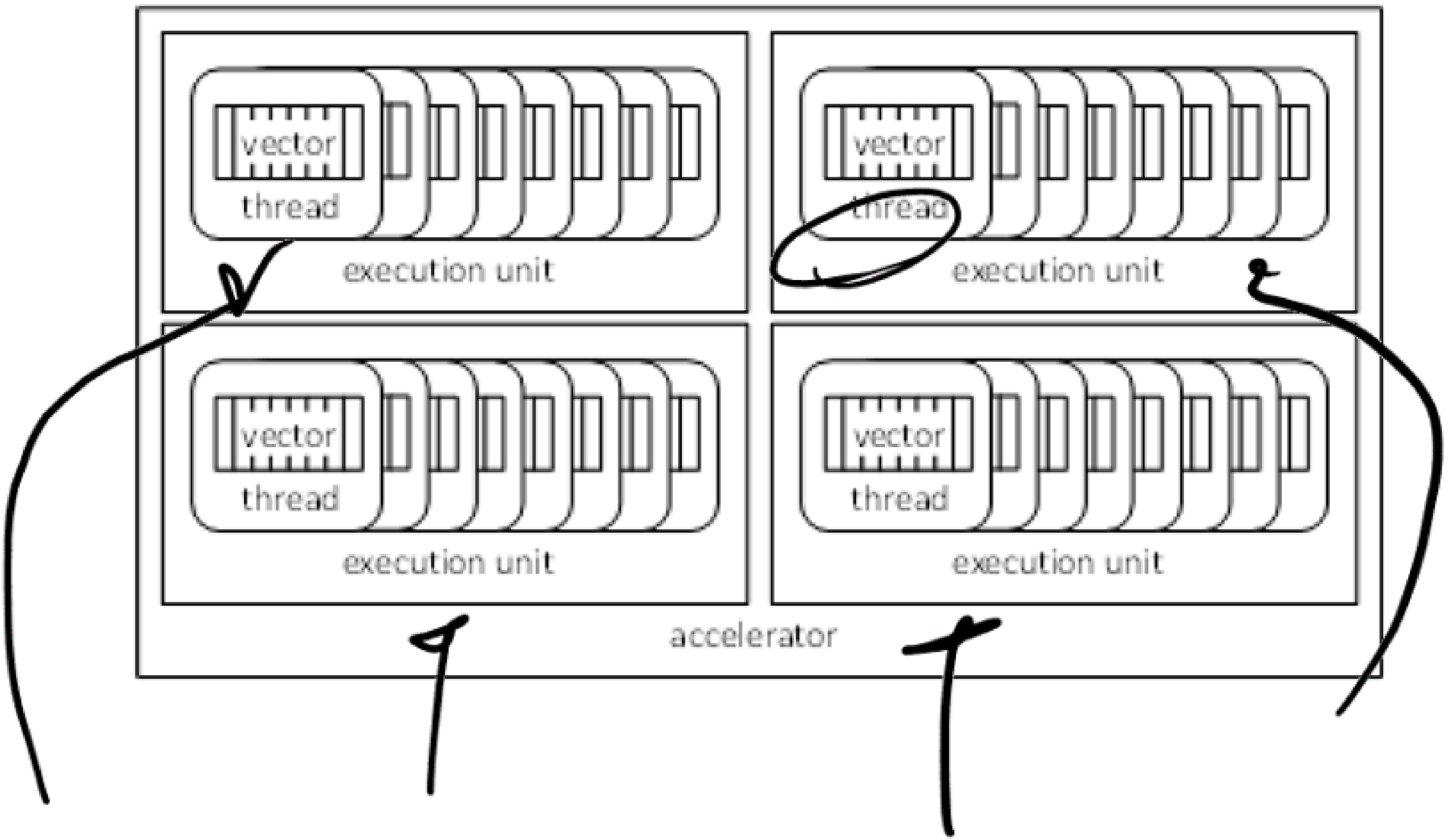
[illegible]



Processing flow on CUDA

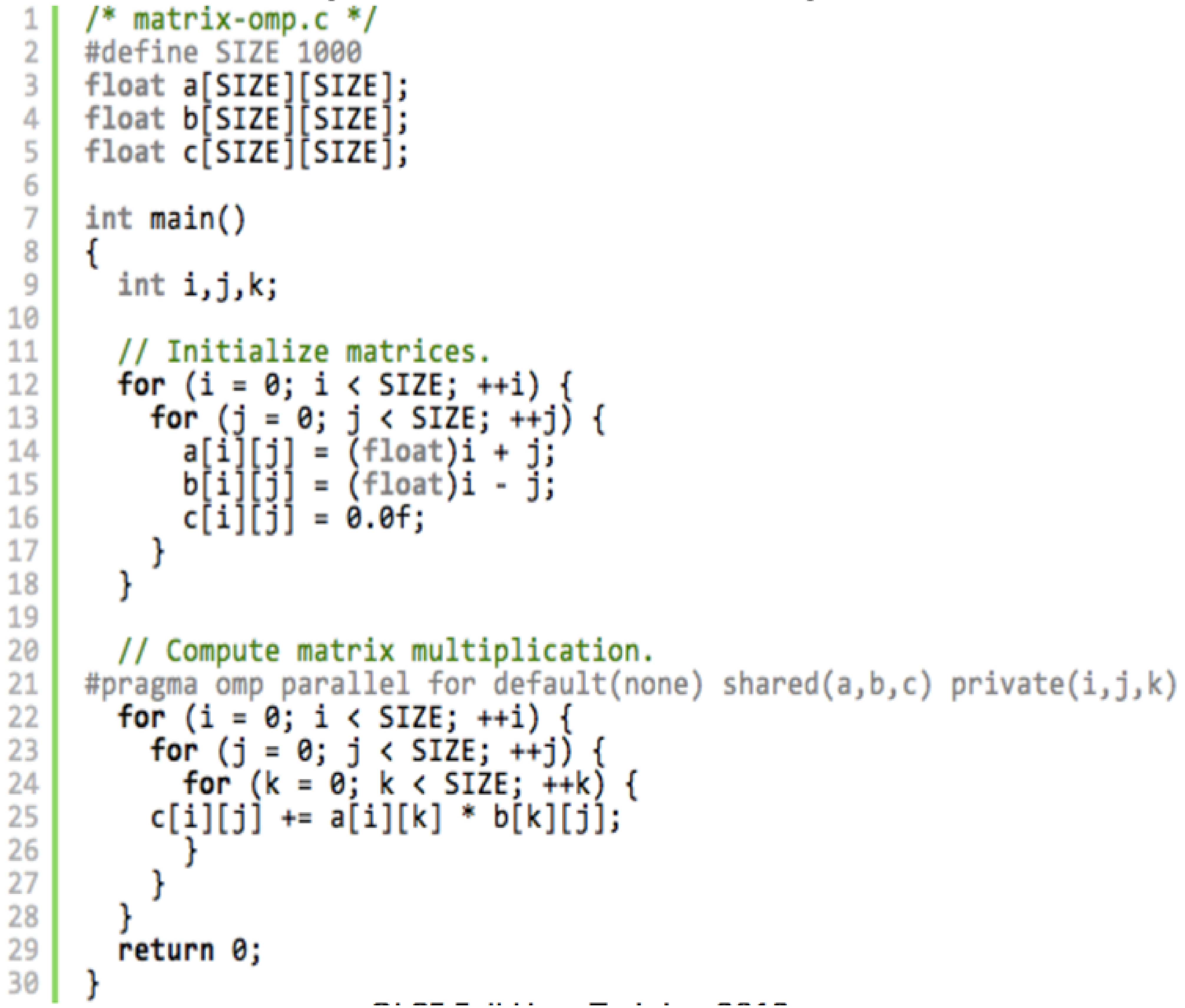
Open ACC Device Model

OpenACC Device Model



OpenMP Example

Example



```c
/* matrix-omp.c */
#define SIZE 1000
float a[SIZE][SIZE];
float b[SIZE][SIZE];
float c[SIZE][SIZE];

int main()
{
  int i,j,k;

  // Initialize matrices.
  for (i = 0; i < SIZE; ++i) {
    for (j = 0; j < SIZE; ++j) {
      a[i][j] = (float)i + j;
      b[i][j] = (float)i - j;
      c[i][j] = 0.0f;
    }
  }

  // Compute matrix multiplication.
#pragma omp parallel for default(none) shared(a,b,c) private(i,j,k)
  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];
      }
    }
  }
  return 0;
}
```

Simple Matrix-Matrix Multiplication in OpenACC

```
void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw)
{

 #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw]) copyout(P[0:Mh*Nw])
 for (int i=0; i<Mh; i++) {
   #pragma acc loop
   for (int j=0; j<Nw; j++) {
     float sum = 0;
     for (int k=0; k<Mw; k++) {
        float a = M[i*Mw+k];
        float b = N[k*Nw+j];
        sum += a*b;
      }
      P[i*Nw+j] = sum;
   }
 }
}
```

Run in accelerator

Some Observations

- The code is almost identical to the sequential version, except for the two lines with #pragma at line 4 and line 6.
- OpenACC uses the compiler directive mechanism to extend the base language.
  - #pragma at line 4 tells the compiler to generate code for the 'i' loop at line 5 through 16 so that the loop iterations are executed in parallel on the accelerator.
  - The copyin clause and the copyout clause specify how the matrix data should be transferred between the host and the accelerator. The #pragma at line 6 instructs the compiler to map the inner 'j' loop to the second level of parallelism on the accelerator.

Data Clauses

- Data clauses -- a comma-separated collection of variable names, array names, or subarray specifications. Compiler allocates and manage a copy of variable or array in device memory, creating a visible device copy of variable or array.
- Data clauses:
  - copy
  - Copyin
  - Copyout
  - Create
  - Present
  - present_or_copy
  - present_or_copyin
  - present_or_copyout
  - present_or_create

Data Clauses

- copy(list)
  - Allocates the data in list on the accelerator and copies the data from the host to the accelerator when entering the region, and copies the data from the accelerator to the host when exiting the region.
- copyin(list) — Nothing on Arrive.
  - Allocates the data in list on the accelerator and copies the data from the host to the accelerator when entering the region.
- copyout(list) — data back from
  - Allocates the data in list on the accelerator and copies the data from the accelerator to the host when exiting the region.
- create(list) Acc
  - Allocates the data in list on the accelerator, but does not copy data between the host and device.

Data Clauses

- present(list)
  - The data in list must be already present on the accelerator, from some containing data region; that accelerator copy is found and used.
- present_or_copy(list)
  - If the data in list is already present on the accelerator from some containing data region, that accelerator copy is used; if it is not present, this behaves like the copy clause.
- present_or_copyin(list)
  - If the data in list is already present on the accelerator from some containing data region, that accelerator copy is used; if it is not present, this behaves like the copyin clause.
- present_or_copyout(list)
  - If the data in list is already present on the accelerator from some containing data region, that accelerator copy is used; if it is not present, this behaves like the copyout clause.
- present_or_create(list)
  - If the data in list is already present on the accelerator from some containing data region, that accelerator copy is used; if it is not present, this behaves like the create clause.
- deviceptr(list)
  - C and C++; the list entries must be pointer variables that contain device addresses, such as from acc_malloc.

Simple Matrix-Matrix Multiplication in OpenACC again

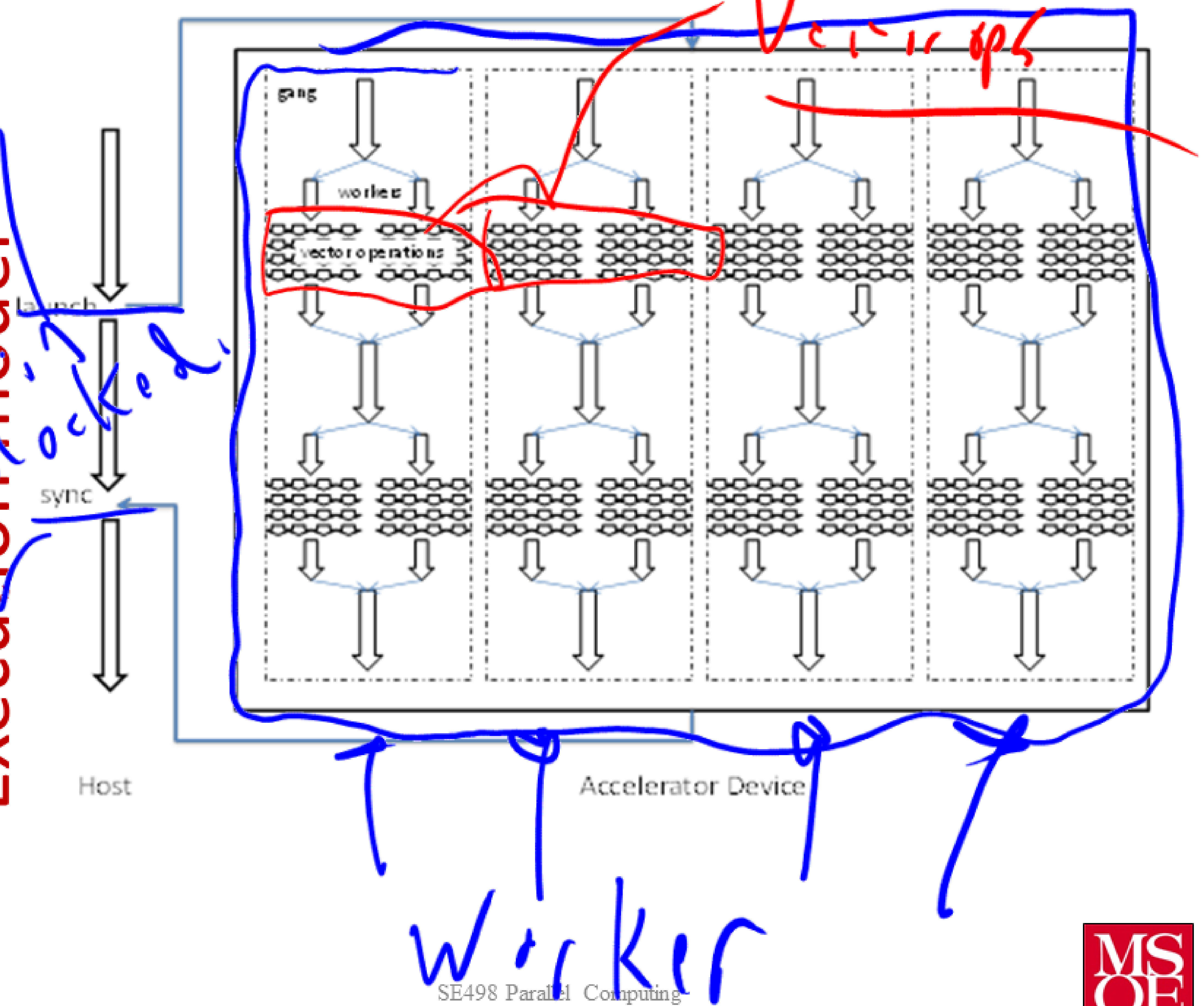
```
void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw)
{

 #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw]) copyout(P[0:Mh*Nw])
 for (int i=0; i<Mh; i++) {
    #pragma acc loop
    for (int j=0; j<Nw; j++) {
       float sum = 0;
       for (int k=0; k<Mw; k++) {
          float a = M[i*Mw+k];
          float b = N[k*Nw+j];
          sum += a*b;
       }
       P[i*Nw+j] = sum;
    }
 }
}
```

Gangs workers and vectors

- OpenACC execution model has three levels:
  - gang
  - worker
  - Vector
- An OpenACC gang is a threadblock
- A worker is effectively a warp
- An OpenACC vector is a form of CUDA thread.

Execution model



Interpreting openACC code

```
#pragma acc parallel num_gangs(32)
{
    Statement 1; Statement 2;
    #pragma acc loop gang
    for (int i=0; i<n; i++) {
       Statement 3; Statement 4;
    }
    Statement 5;  Statement 6;
    #pragma acc loop gang
    for (int i=0; i<m; i++) {
       Statement 7;  Statement 8;
    }
    Statement 9;
    if (condition)
      Statement 10;
}
```

- Statements 1 and 2 are redundantly executed by 32 gangs
- The n for-loop iterations are distributed to 32 gangs

Also executed multiple times

Distributed to 32 gangs.

Done

Example

In this example, number of gangs and workers in each gang specified in clauses, otherwise decided by compiler.

```
#pragma acc parallel loop copyin(a, b), copyout (c),
num_gangs(1024) num_workers(32)
{
        c = b + a;
        …
}
```

One worker in each gang (gang leader) executes statement, so this statement repeated by each gang leader, so this would not make much sense in itself.

loop construct

```
#pragma acc loop [clause] ...
        for loop
```

Loop directive is used within a parallel directive and the two can be combined:

```
#pragma acc parallel loop [clause] ...
        for loop
```

Loop is parallelized. With no gang, worker or vector clauses, implementation can automatically select whether to execute loop across gangs, workers within a gang, or whether to execute as vector operations.

Implementation may choose to use vector operations to execute any loop with no loop directive, using classical automatic vectorization."

Example
Matrix multiplication

```
void matrixMultACC(float *C, const float *A, const float *B, int N)
{
        #pragma acc parallel loop copyin(A[0:N*N], B[0:N*N]) copyout (C[0:N*
        for (int I = 0; i < N; i++) {
                #pragma acc loop
                for (int j = 0; j < N; j++) {
                        float sum = 0;
                        for (int k = 0; k < N; k++) {
                                sum += A[i*N + k] * B[k*N + j];
                        }
                        C[i*N + j] = sum;
                }
        }
}
```

Subarray locations 0 to N*N - 1

In this example left to compiler to decide how to allow resources to parallelize loop

Question

loop

In the following how is the following executed?

```
#pragma acc parallel num_gangs(1024)
{
        for (int i = 0; i < 2048; i++) {
        …
        }
}
```

Redundant

1024 X

Each of the 1024 gang leaders will execute the same for loop as a normal sequential for loop (redundantly)!!!

gang and worker clauses

In an accelerator parallel region:

gang clause specifies that the iterations of the associated loop or loops are to be executed in parallel by distributing the iterations among the gangs created by the parallel construct. The loop iterations must be data independent, except for variables specified in a reduction clause.

worker clause specifies that the iterations of the associated loop or loops are to be executed in parallel by distributing the iterations among the multiple workers within a single gang. The loop iterations must be data independent, except for variables specified in a reduction clause.

gang Example

```
#pragma acc parallel num_gangs(1024)
{
          pragma acc loop gang
          for (int i = 0; i < 2048; i++) {
          …
          }
}
```

Now the 2048 iterations are shared among the gang leaders, i.e. two iterations each.

gang and worker Example

```
#pragma acc parallel num_gangs(1024) num_workers(32)
{
        pragma acc loop gang
        for (int i = 0; i < 2048; i++) {
                pragma acc loop worker
                for (int j = 0; j < 512; j++) {
                        foo(i,j);
                }
        }
}
```

foo(i,j) will be executed 2048 x 512 times, distributed across 1024 x 32 workers.

Each gang will execute two iterations of the outer loop (2048/1024 = 2)

For each outer loop iteration, each worker in a gang will execute 16 iterations of the inner loop (512/32 = 16)

This assume even distribution which the spec. does not actually say.

Code from page 323, Chapter 15 of Programming Massively Parallel Processors by D. B. Kirk and W-M W. Hwu, Morgan Kaufmann, 2013.

Question

Analyze the following code and explain how it gets executed.

How could this be written in CUDA?

```
#pragma acc parallel num_gangs(32)
{
        Statement 1;
        Statement 2;
        #pragma acc loop gang
        for (int i = 0; i < n; i++) {
                Statement 3;
                Statement 4;
        }
        Statement 5;
        Statement 6;
        #pragma acc loop gang
        for (int i = 0; i < n; i++) {
                Statement 7;
                Statement 8;
        }
        Statement 9;
        if (condition)
                Statement 10;
}
```

vector clause

vector clause specifies that loop iterations are executed in vector or SIMD mode.

Using vectors of length specified or chosen for parallel region.

Implementation-defined whether a loop with **vector** clause may contain a loop containing **gang** or **worker** clause.

```
#pragma acc parallel num_gangs(1024)
num_workers(32) vector_length(32)
{
        pragma acc loop gang
        for (int i = 0; i < 2048; i++) {
                pragma acc loop worker
                for (int j = 0; j < 512; j++) {
                        pragma acc loop
vector
                        for (int k = 0; k <
512; k++) {

                        }
                }
        }
}
```

Possibly mapping
Gang to CUDA block
Worker to CUDA warp
Vector element to thread within warp

Kernel construct

#pragma acc kernels *[clause]* ...
structured block

Defines a region of the program that is to be compiled into a sequence of kernels for execution on the accelerator device.

Compiler breaks code in the kernels region into a sequence of accelerator kernels.

Typically, each loop nest will be a distinct kernel.

When program encounters a kernels construct, it will launch sequence of kernels in order on the device. Number and configuration of gangs of workers and vector length may be different for each kernel.

Example

```
#pragma acc kernels
{
        #pragma acc loop num_gangs(1024)
        for (int i = 0; i < 2048; i++) {
                a[i] = b[i];
        }
        #pragma acc loop num_gangs(512)
        for (int j = 0; j < 2048; j++) {
                c[j] = a[j]*2;
        }
        for (int k = 0; k < 2048; k++) {
                d[k] = c[k];
        }
}
```

Each kernel can have different gang number/size

The loop construct says share loop iterations among gang leaders only

As opposed to the parallel construct where gang leader redundantly execute this loop, here a single for loop executed as a kernel with 2048 iterations

Data Dependencies

```
void foo(int *x, int *y, int n, int m) {
        int a[2048], b[2048];
        #pragma acc kernels copy(x[0;2048], y[0:2048], a, b)
        {
        #pragma acc loop
        for (int i = 0; i < 2047; i++) {
                a[i] = b[I + 1] + 1;
        }
        #pragma acc loop
        for (int j = 0; j < 2047; j++) {
                a[j] = a[j + 1] + 1;
        }
        #pragma acc loop
        for (int k = 0; k < 2047; k++) {
                x[k] = y[k + 1] + 1;
        }
        #pragma acc loop
        for (int l = 0; l < m; l++) {
                x[l] = y[l + n] + 1;
        }
}
```

No data dependence

Data dependence

No data dependence if x[] not aliased with y[]

No data dependence if n >= m

Example - COnvolution

```
void convolution_SM_N(typeToUse A[M][N], typeToUse B[M][N])
{
  int i, j, k;
  int m=M, n=N;
  // OpenACC kernel region
  // Define a region of the program to be compiled into a sequence of
  // kernels for execution on the accelerator device
  #pragma acc kernels pcopyin(A[0:m]) pcopy(B[0:m])
  {
    typeToUse c11, c12, c13, c21, c22, c23, c31, c32, c33;
    c11 = +2.0f; c21 = +5.0f; c31 = -8.0f;
    c12 = -3.0f; c22 = +6.0f; c32 = -9.0f;
    c13 = +4.0f; c23 = +7.0f; c33 = +10.0f;
    // The OpenACC loop gang clause tells the compiler that the iterations
    // of the loops are to be executed in parallel across the gangs.
    // The argument specifies how many gangs to use to execute the
    // iterations of this loop.
    #pragma acc loop gang(64)
    for (int i = 1; i < M - 1; ++i)
    {
      // The OpenACC loop worker clause specifies that the iteration
      // of the associated loop are to be
      // executed in parallel across the workers within the gangs created.
      // The argument specifies how many workers to use to execute the
      // iterations of this loop.
      #pragma acc loop worker(128)
      for (int j = 1; j < N - 1; ++j)
      {
        B[i][j] = c11 * A[i - 1][j - 1] + c12 * A[i + 0][j - 1] + c13 * A[i + 1][j - 1]
        + c21 * A[i - 1][j + 0] + c22 * A[i + 0][j + 0] + c23 * A[i + 1][j + 0]
        + c31 * A[i - 1][j + 1] + c32 * A[i + 0][j + 1] + c33 * A[i + 1][j + 1];
      }
    }
  } //kernels region
}
```

In Class activity

- Lets convert our image convolution to use openAcc and see if it works better.

Advantages of OpenACC

- OpenACC programmers can often start with writing a sequential version and then annotate their sequential program with OpenACC directives.
  - leave most of the details in generating a kernel and data transfers to the OpenACC compiler.
- OpenACC code can be compiled by non-OpenACC compilers by ignoring the pragmas.

Problems with OpenACC

- The main trouble with OpenACC is data movement
  - Data movement takes up a good deal of the execution time and should be considered when deciding if OpenACC is the correct route to go.
- Some OpenACC pragmas are hints to the OpenACC compiler, which may or may not be able to act accordingly
  - The performance of an OpenACC depends heavily on the quality of the compiler.
  - Much less so in CUDA or OpenCL
- Some OpenACC programs may behave differently or even incorrectly if pragmas are ignored