Codee Training Series

April 26-27, 2022

NERSC

codee

Shift Left Performance

Automated Code inspection for Performance

Second: Addressing GPU challenges with Codee

#2 Usage of Codee for GPU programming (1/2)

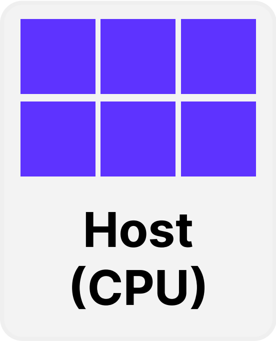
- The **GPU programming challenges**
- Memory usage, massive parallelism exploitation, and data transfers minimization
- **Codee's support** to **find opportunities for offloading** and **optimize memory layout for data transfers**
- Hands-on: Optimizing **MATMUL** on Perlmutter

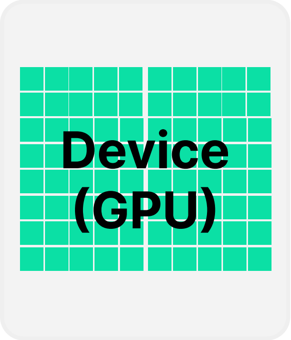
Format:

- Remote lectures (~30'), demos, and hands-on sessions

What are the differences between CPUs and GPUs?

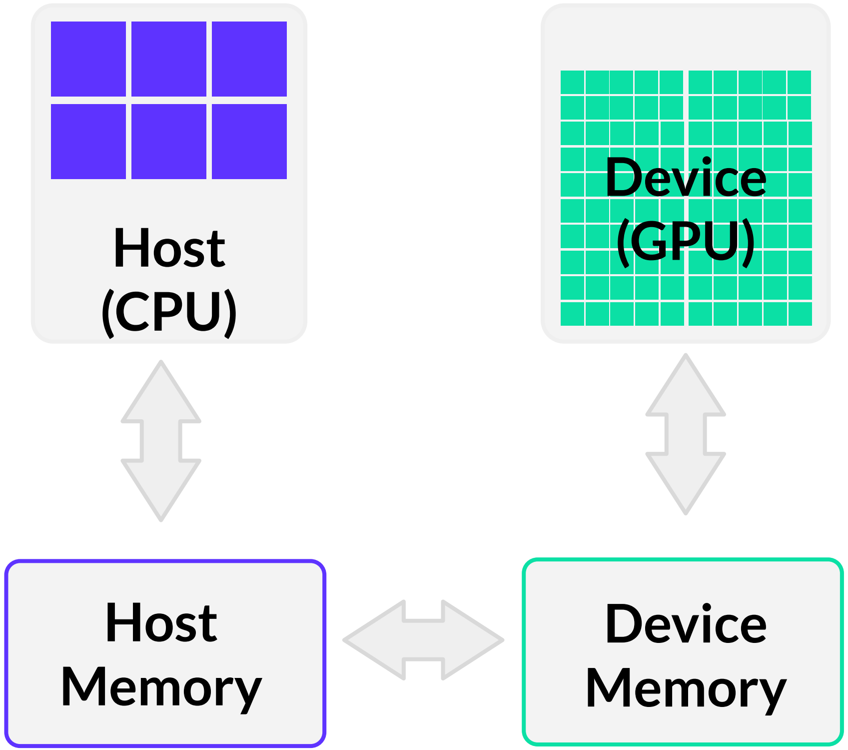
- **First, the number of cores available in the hardware**
  - GPUs have many many more cores than CPUs
- **Second, the grouping of the threads at the hardware level**
  - In CPUs, the threads are not grouped and all the threads are executed at the same time
  - In GPUs, the threads are grouped and all the threads in a group are executed at the same time.
- **Third, the complexity of the memory design**
  - In CPUs, all the threads access to all the memory
  - In GPUs, there are constraints in the memory that can be accessed by the threads (e.g., cache, texture, scratchpad, global).
- **Fourth, execution of instructions in vector mode**
  - Both CPUs and GPUs exploit vector processing, although different “flavours” of it.





The GPU Execution Model

- **Use of a host-driven execution model.**
- **Sequential code runs on a conventional processor.**
- **Computationally intensive parallel pieces of code (kernels) run on an accelerator such as a GPU.**
- **To maximize performance, high-performance applications generally conform to the following three rules of accelerator programming:**
    - Transfer the data onto the device and keep it there.
    - Give the device enough work to do.
    - Focus on data reuse within the device(s) to avoid memory bandwidth bottlenecks



The GPU programming challenges: Example codes...

		Challenges of GPU acceleration addressed in introductory course			Other GPU programming challenges to be addressed in next advanced course			
		Find opportunities for offloading	Optimize memory layout for data transfers	Identify defects in data transfers	Exploit massive parallelism through loop nest collapsing	Minimize data transfers across consecutive loop nests	Minimize data transfers through convergence loops	Identify auxiliary functions to be offloaded
Example codes used in this introductory course	**PI**	x	-	-	-	-	-	-
	MATMUL	x	x	x	x	x	-	-
	LULESHmk	x	x	x	x	x	x	x
	HEAT	x	-	-	-	x	x	-
	Your code!	Probably all of these challenges apply, and even more!						

The GPU Programming Challenges in this Introductory Course

Challenge #1: Find opportunities for offloading

- **Code patterns: computation patterns (eg. loops will execute correctly on the GPU)**
- On GPUs: Start offloading computations to the GPU, guaranteed correctness!
- On CPUs: Usually the same code analysis is required to execute the computations in parallel correctly!

Challenge #2: Optimize memory layout for data transfers

- **Code patterns: memory patterns (eg. shaping arrays)**
- On GPUs: Watch your data structure design as it may break your code!
- On CPUs: Hardware keeps memory consistency, so focus mostly on locality!

Challenge #3: Identify defects in data transfers

- **Code patterns: computation and memory patterns (eg. deep copy)**
- On GPUs: Data transfers for complex data structs are often not managed automatically!
- On CPUs: Often not a big issue as there is shared memory!

Why using additional tools apart from APIs?

- **The OpenACC Application Programming Interface. Version 2.7 (November 2018)** 🔗
  - “does **not describe automatic detection of parallel regions or automatic offloading of regions of code to an accelerator by a compiler or other tool**.”
  - “if one thread updates a memory location and another reads the same location, or two threads store a value to the same location, **the hardware may not guarantee the same result** for each execution.”
  - “it is (...) **possible to write a compute region that produces inconsistent numerical results**.”
  - “**Programmers need to be very careful that the program uses appropriate synchronization** to ensure that an assignment or modification by a thread on any device to data in shared memory is complete and available before that data is used by another thread on the same or another device.”
- **Programmers are responsible for making good use of Application Programming Interface (API)**
  - This applies to OpenACC, OpenMP
  - But also to any other API, such as MPI, compiler pragmas, and even the programming language itself

Shaping Arrays in OpenMP/OpenACC

- **Provide the compiler with information about array size and array ranges.**
- **Helps the compiler ensure correct memory allocation on the device**
- **Add the shape specification to the data clauses**, e.g.:

```
x[start:count]
```

where `start` is the first element to be copied and `count` is the number of elements to copy.

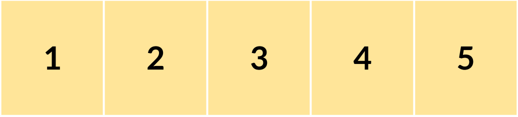
- **Allows storing of only part of the array on the device**

```
#pragma acc data create(x[0:N]) copyout(y[0:N])

!$acc data create(x(0:N)) copyout(y(0:N))
```

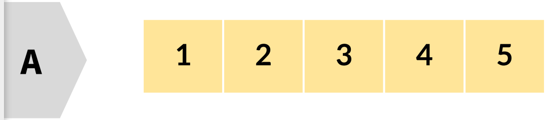
Shaping Arrays 1D in OpenMP/OpenACC

- **Vectors are typically implemented as arrays 1D.**
- **Developer can choose between static and dynamic memory allocation.**
  - Static arrays are allocated on the stack, which is limited.
  - As a result, large arrays can make the application crash.
- **Actual data is stored in consecutive memory locations, which triggers compiler optimizations.**

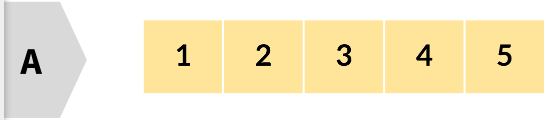


VECTOR size 5

```
double *A = malloc(...)
for(i) {
  … A[i] …
}
```




```
double A[9]
for(i) {
  … A[i] …
}
```



Shaping Arrays 2D in OpenMP/OpenACC

- **Matrices are typically implemented as “arrays 2D”, but what is the actual memory layout?**
  - It depends on the programming language: row-major in C/C++ and column-major in Fortran.
- **Developer can choose between static and dynamic memory allocation.**
- **Actual data MAY NOT be stored in consecutive memory locations, disabling compiler optimizations.**

1	2	3
4	5	0
0	6	0

MATRIX 3x3

Shaping Arrays 2D in OpenMP/OpenACC

- **Statically allocated arrays** guarantee that actual data is stored in consecutive memory locations. Note that C/C++ and Fortran defer in the order of the data inside de consecutive memory region.

1	2	3
4	5	0
0	6	0

MATRIX 3x3

```
double A[3][3]
for(i) {
  for(j) {
    … A[i][j] …
  }
}
```

A

1	2	3	4	5	0	0	6	0

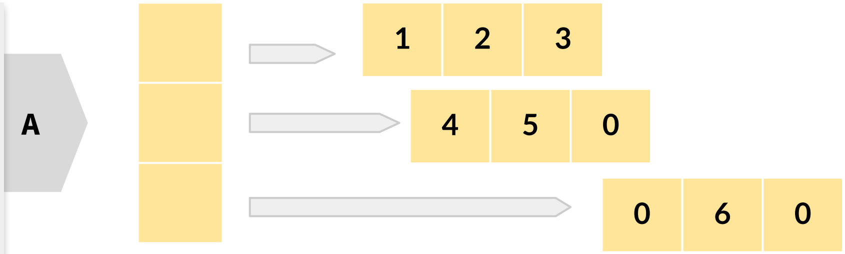
Shaping Arrays 2D in OpenMP/OpenACC

1	2	3
4	5	0
0	6	0

MATRIX 3x3

- **Dynamically allocated arrays** are controlled by the programmer, who is responsible for memory allocation and initialization. How can the programmer guarantee consecutive memory allocation?
- **Dynamically allocated arrays without consecutive memory:**

```
double **A = malloc(3)
for(i) {
  A[i] = malloc(3)
}
for(i) {
  for(j) {
    … A[i][j] …
  }
}
```



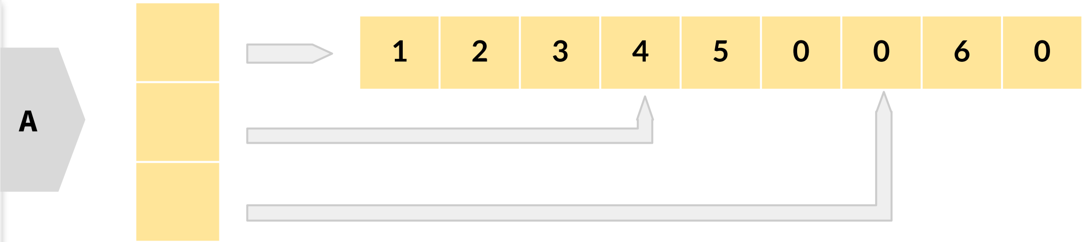
Shaping Arrays 2D in OpenMP/OpenACC

- **Dynamically allocated arrays with consecutive memory:** Several options are possible…
- Option 1:
  - Data elements are stored in one single consecutive array
  - An auxiliary pointer-type array is used to facilitate access to each row
  - This enables array accesses through the common notation A[i][j]

1	2	3
4	5	0
0	6	0

MATRIX 3x3

```
double **A = malloc(3)
double *Aaux = malloc(3x3)
for(i) {
  A[i] = Aaux + i * 3
}
for(i) {
  for(j) {
    … A[i][j] …
  }
}
```



In this design, the programmer needs to allocate two separate arrays and initialize the pointers as offset with respect to the beginning of the consecutive memory region.

Shaping Arrays 2D in OpenMP/OpenACC

- **Many scientific codes use an alternative data structure**
- Option 2:
  - Data elements are stored in one single consecutive array
  - Rewrite all the 2D array accesses as 1D array accesses
  - Typically A[i][j] is rewritten as A[i*N+j]

1	2	3
4	5	0
0	6	0

MATRIX 3x3

```
double *A = malloc(3x3)
for(i) {
  for(j) {
    … A[i*3+j] …
  }
}
```

A

1	2	3	4	5	0	0	6	0

And in this design, the programmer minimizes the memory consumption but must change all of the accesses in the code.

Shaping Arrays 2D in OpenMP/OpenACC

- **Dynamically allocated arrays for sparse matrices:** Sparse storage format uses several auxiliary arrays to avoid storing the elements with value equal to zero.
- For example: Compressed Row Storage (CRS) format

1	2	3
4	5	0
0	6	0

MATRIX 3x3

```
double *A
int *rowIdx
int *colIdx
```

```
for(i) {
  for(j=rowIdx(i),rowIdx(i+1)-1) {
    … A[j] …
  }
}
```

A	1	2	3	4	5	6
colIdx	0	1	2	0	1	1
rowIdx	0	3	5	6		

How array shaping affects in OpenMP/OpenACC?

- Array shaping in OpenMP/OpenACC affects to **how to code data transfers**.
- OpenMP/OpenACC clauses that tell the compiler **what** data must be transferred between CPU memory and GPU memory and **how**.
- MATMUL example code using **double** data type**.

```
// C (m x n) = A (m x p) * B (p x n)
void matmul(size_t m, size_t n, size_t p, double **A, double **B, double **C) {
. . .
    // Accumulation
    #pragma acc data copyin(A[0:m][0:p], B[0:p][0:n], m, n, p) copy(C[0:m][0:n])
    {
    #pragma acc parallel
    {
    #pragma acc loop
    for (size_t i = 0; i < m; i++) {
        for (size_t j = 0; j < n; j++) {
            for (size_t k = 0; k < p; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    } // end parallel
    } // end data
}
```

```
. . .

// C (m x n) = A (m x p) * B (p x n)
void matmul(size_t m, size_t n, size_t p, double **A, double **B, double **C) {
. . .
    // Accumulation
    #pragma omp target enter data map(to: A[0:m])
    for(int i0 = 0; i0 < m; ++i0) {
      #pragma omp target enter data map(to: A[i0][0:p])
    }
    #pragma omp target enter data map(to: B[0:p])
    for(int i0 = 0; i0 < p; ++i0) {
      #pragma omp target enter data map(to: B[i0][0:n])
    }
    #pragma omp target enter data map(to: C[0:m])
    for(int i0 = 0; i0 < m; ++i0) {
      #pragma omp target enter data map(to: C[i0][0:n])
    }
    #pragma omp target teams distribute parallel for shared(A, B, m, n, p) map(to: m, n, p) schedule(static)
    for (size_t i = 0; i < m; i++) {
        for (size_t j = 0; j < n; j++) {
            for (size_t k = 0; k < p; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    for(int i0 = 0; i0 < m; ++i0) {
      #pragma omp target exit data map(from: C[i0][0:n])
    }
    #pragma omp target exit data map(from: C[0:m])
}
. . .
```

www.codee.com
info@codee.com
Subscribe: codee.com/newsletter/
USA - Spain
codee_com
company/codee-com/