Parallelware Tool Workshop

Learning parallelization of real applications from the ground-up

Manuel Arenaz | October 17, 2019

©Appentra Solutions S.L.

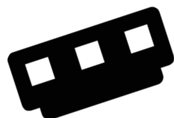
Agenda

8:15 - 8:45	*Morning refreshment and coffee*
8:45 - 9:00	*Welcome and introductions*
9:00 - 9:30	Lecture 1: An introduction to OpenMP/OpenACC optimizations for CPUs/GPUs
9:30 - 10:15	Lecture 2: Patterns to minimize data transfers, optimize memory usage and exploit massive parallelism
10:15 - 10:30	Break
10:30 - 11:00	Lecture 3: Minimizing data transfers
11:00 - 11:30	**Lecture 4: Optimizing memory usage**
11:30 - 12:00	Lecture 5: Exploiting massive parallelism
12:00 - 13:00	Working lunch (hands-on activities)
13:00 - 14:00	Practical 5A: Parallelizing the calculation of HEAT
14:00 - 17:00	Hands-on time with your code
17:00 pm	*Close*

Use cases: Performance optimization on CPU/GPU

Use case #1: Minimizing data transfers

- **Code patterns: flow patterns (eg. convergence loop)**
- On GPUs: Transfer data from CPU to GPU and reuse it!
- On CPUs: Create threads and reuse them!



Use case #2: Optimizing memory usage

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

Use case #3: Exploiting massive parallelism

- **Code patterns: computation patterns (eg. collapsible nested loops)**
- On GPUs: Scale-up to thousands of threads!
- On CPUs: Limited number of threads, so not so important as on GPUs!

Why using memory patterns?

1: Memory patterns enable to exploit locality during the execution of parallel code

- By exploiting data locality in a parallel code, each thread will have very fast access to the data needed for the computations; and this is key for performance in modern hardware systems (e.g. stride one memory access).

2: Memory patterns provide rules to re-code the data structure of the variables

- The data structure of the variables dictates the memory layout of the data of our programs.
- It must be designed with memory layout in mind (e.g. AoS -Struct of Arrays- not AoS -Array of Structs).

3: Memory patterns enable to the detection of errors/bugs in the parallel code

- Data structures typically allocate memory not only for the actual data, but also for auxiliary data structures that contain pointers to enable seamless access to the actual data.
- It may lead to incorrect memory accesses when data needs to be moved around different memory systems, as it is the case of moving data between CPU memory and GPU memory.

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))
```

Memory Patterns

data structure design patterns

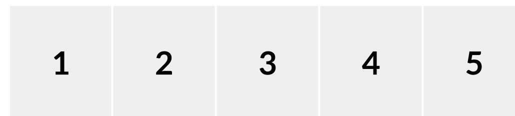
(e.g. array 1D, multi-dimensional array, array-of-structs/struct-of-arrays)

data access patterns

(e.g. linear, strided, irregular, stencil)

Shaping Arrays 1D in OpenMP/OpenACC

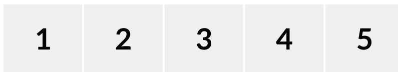
- Vectors are typically implemented as arrays 1D.
- Developer can choose between static and dynamic memory allocation.
  - Static arrays are allocated on the stack, which is limited.
  - As a result, large arrays can make the application crash.
- Actual data is stored in consecutive memory locations,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



VECTOR size 5

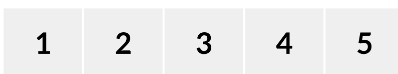
```
double *A = malloc(...)
for(i) {
  … A[i] …
}
```

A




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

A



appentra
make code parallel

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

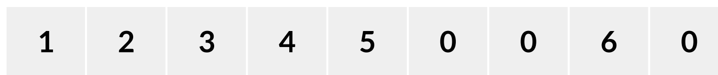
MATRIX 3x3

appentra
make code parallel

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.

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

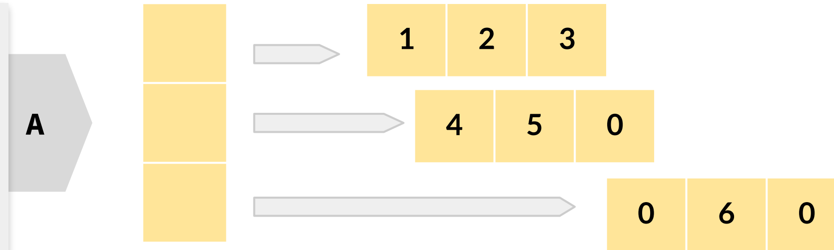


appentra
make code parallel

Shaping Arrays 2D in OpenMP/OpenACC

- **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] …
  }
}
```



 ©Appentra Solutions S.L.

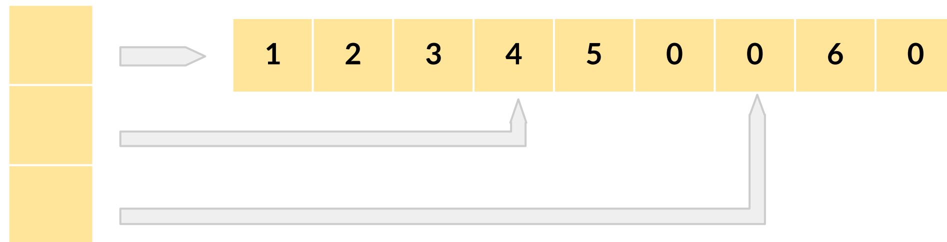
appentra
make code parallel

Shaping Arrays 2D in OpenMP/OpenACC

- **Dynamically allocated arrays with consecutive memory**: Several options...

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

A



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.

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

A

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.

```
double *A
int *rawIdx
int *cdIdx
```

```
for(ij) {
  … A[rawIdx(ij)*3+colIdx(ij)] …
}
```

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

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

A	1	2	3	4	5	6
colIdx	0	1	2	0	1	1
rawIdx	0	3	5	7		

 ©Appentra Solutions S.L.

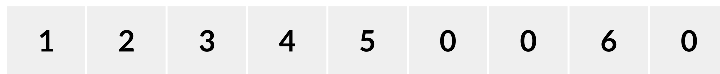
appentra
make code parallel

How array shaping affects in OpenMP/OpenACC?

- Array shaping in OpenMP/OpenACC clauses tells the compiler what data must be transferred between CPU memory and GPU memory.

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

A

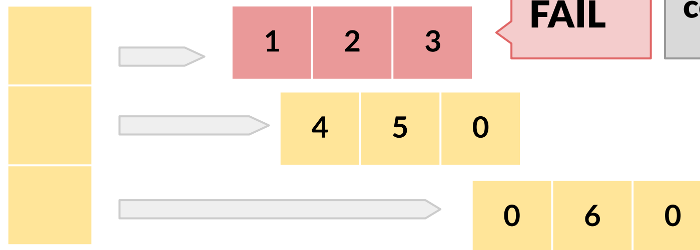


OK

copy(A[0:3][0:3])

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

A



FAIL

copy(A[0:3][0:3])

Parallelizing MATrix MULtiplication on the GPU with OpenMP/OpenACC

Walkthrough:

- Using Parallelware Trainer in file version *matmul_v1*, function *matmul()*, first loop:
  - Generate OpenACC code, remove the array shape of C in *copyout* clause, and watch the defect.
- Using Parallelware Trainer in file version *matmul_v2*, function *matmul()*, first loop:
  - Generate OpenMP code, remove the array shape of C in *map* clause, and watch the defect.
- Using Parallelware Trainer in file version *matmul_v3*, function *matmul()*, first loop:
  - Generate OpenACC code, remove the (incomplete) array shape of C in *map* clause, and watch the defect.