Codee Training Series
April 26-27, 2022

Shift Left Performance
Automated Code inspection for Performance
Third: Addressing more GPU challenges with Codee

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

- The **GPU programming challenges**
- Codee’s support to identify defects in data transfers
- Hands-on: **Optimizing MATMUL** on Perlmutter

Format: sessions
- Remote lectures (~30’), demos, and hands-on exercises
Performance Optimization Platform

Opportunities (OPP)
Sequential, vectorization, multi-threading and GPU offloading

Recommendations (PWR)
Boost performance and ensure best practices

Defects (PWD)
Find and fix bugs in parallel code and correctness verification

Remarks (RMK)
Proficient usage of tools

Scan source code without executing that code

Report human-readable actionable recommendations on where and how to fix performance issues

Compliance with performance optimization best practices (memory usage, vectorization, multi-threading, offload)

Optimize performance for microprocessors (x86, Arm, Power) and accelerators (GPU)

Automated fixes to actually implement code changes

Customization and extension of built-in rule set

Full workflow support: CI/CD, repository, IDE and issue trackers

example/matmul$ pwreport src/main.c:15 --level 2 -- -I src/include
Compiler flags: -I src/include

FUNCTION BEGIN at src/main.c:matmul:6:1
6: void matmul(size_t m, size_t n, size_t p, double **A, double **B, double **C) {
LOOP BEGIN at src/main.c:matmul:15:5
15: for (size_t i = 0; i < m; i++) {
[PWR010] src/main.c:15:5 'B' multi-dimensional array not accessed in row-major order
[RMK005] src/main.c:18:28 avoid non-consecutive array access for variable 'A' to improve performance
[RMK005] src/main.c:18:38 avoid non-consecutive array access for variable 'B' to improve performance
[RMK005] src/main.c:18:25 avoid non-consecutive array access for variable 'C' to improve performance
[OPP001] src/main.c:15:5 is a multi-threading opportunity
[OPP003] src/main.c:15:5 is a offload opportunity
LOOP END
FUNCTION END
FUNCTION BEGIN at src/main.c:main:24:1
24: int main(int argc, char *argv[]) {
FUNCTION END
Open Catalog of Coding Rules for Performance

https://www.codeee.com/knowledge/

Recommendations (40)

PWR001: Declare global variables as function parameters
PWR002: Declare scalar variables in the smallest possible scope
PWR003: Explicitly declare pure functions
PWR004: Declare OpenMP scoping for all variables

Opportunities (3)

OPP001: Multi-threading opportunity
OPP002: SIMD opportunity
OPP003: Offloading opportunity

Defects (11)

PWD002: Unprotected multithreading reduction operation
PWD003: Missing array range in data copy to the GPU
PWD004: Out-of-memory-bounds array access
PWD005: Array range copied to or from the GPU does not cover the used range

Remarks (14)

RMK001: Loop nesting that might benefit from hybrid parallelization using multithreading and SIMD
RMK002: Loop nesting that might benefit from hybrid parallelization using offloading and SIMD
RMK003: Potentially privatizable temporary variable

Glossary (22)

Locality of Reference
Loop fission
Loop interchange
Loop sectioning
Loop tiling
Loop unswitching
Loop-carried dependencies
Memory access pattern
Multithreading
Offloading
## Open Catalog of Coding Rules for Performance: Defects

https://www.codee.com/knowledge/

<table>
<thead>
<tr>
<th>Sequential optimizations</th>
<th>SIMD/Vector execution</th>
<th>Multi-threaded execution</th>
<th>Offloading to accelerators</th>
</tr>
</thead>
<tbody>
<tr>
<td>PWR001: Declare global variables as function parameters</td>
<td>PWR017: Transform while into for loop in order to allow vectorization</td>
<td>PWR006: Avoid privatization of read-only variables</td>
<td>PWR009: Use OpenMP teams to offload work to GPU</td>
</tr>
<tr>
<td>PWR002: Declare scalar variables in the smallest possible scope</td>
<td>PWR018: Call to recursive function within a loop may inhibit vectorization</td>
<td>PWR001: Invalid OpenMP multithreading datascoping</td>
<td>PWR013: Avoid copying unused variables to the GPU</td>
</tr>
<tr>
<td>PWR003: Explicitly declare pure functions</td>
<td>PWR019: Consider interchanging loops to favor vectorization by maximizing inner loop's trip count</td>
<td>PWR002: Unprotected multithreading reduction operation</td>
<td>PWR015: Avoid copying unnecessary array elements to or from the GPU</td>
</tr>
<tr>
<td>PWR004: Declare OpenMP scoping for all variables</td>
<td>PWR020: Consider loop fission to enable vectorization</td>
<td>PWR004: Out-of-memory-bounds array access</td>
<td>PWR024: Loop can be rewritten in OpenMP canonical form</td>
</tr>
<tr>
<td>PWR007: Disable implicit declaration of variables</td>
<td>PWR021: Temporary computation can be extracted to a vectorizable loop</td>
<td>PWR007: Unprotected multithreading recurrence due to out-of-dimension-bounds array access</td>
<td>PWR025: Consider annotating pure function with OpenMP ‘declare simd’</td>
</tr>
<tr>
<td>PWR008: Declare the intent for each procedure parameter</td>
<td>PWR022: Move invariant conditional out of the loop to facilitate vectorization</td>
<td>PWR008: Incorrect privatization in OpenMP parallel region</td>
<td>PWR026: Annotate function for OpenMP offload</td>
</tr>
<tr>
<td>PWR010: Avoid column-major array access in C/C++</td>
<td>PWR023: Add ‘restrict’ for pointer function parameters to hint the compiler that vectorization is safe</td>
<td>PWR010: Incorrect sharing in OpenMP parallel region</td>
<td>PWR027: Annotate function for OpenACC offload</td>
</tr>
<tr>
<td>PWR012: Pass only required fields from derived data types as parameters</td>
<td>PWR021: Temporary computation can be extracted to a vectorizable loop</td>
<td>PWR011: Missing OpenMP last private clause</td>
<td></td>
</tr>
<tr>
<td>RMK004: Avoid strided array access to improve performance</td>
<td>PWR022: Move invariant conditional out of the loop to facilitate vectorization</td>
<td>RMK003: Potential temporary variable for the loop which might be privatizable, thus enabling the loop parallelization</td>
<td></td>
</tr>
<tr>
<td>RMK005: Avoid non-consecutive array access to improve performance</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RMK006: Avoid indirect array access to improve performance</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

https://www.codee.com/knowledge/
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, 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 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.

MATRICES 3x3

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

**MATRIX 3x3**
How array shaping affects in OpenMP/OpenACC?

- Array shaping in OpenMP/OpenACC affects to how to code data transfers.
- And it also affects the correctness of the OpenMP/OpenACC code if the data layout is not managed properly by the programmer (explicitly).

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

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