P. Koumoutsakos
S. Martin
ETH Zentrum, CLT E 13
CH-8092 Zürich

Fall semester 2020

Set 4 - Vectorization and MPI (Part I)

Issued: November 13, 2020
Hand in (optional): November 27, 2020 08:00am

Question 1: Manual Vectorization of Reduction Operator (28 points)

We are interested in optimizing the performance of the following compute kernel

$$r = \sum_{i=1}^{n} a_i, \tag{1}$$

where a_i are the elements of a vector $\boldsymbol{a} \in \mathbb{R}^n$ and $r \in \mathbb{R}$ is the result of reducing $\boldsymbol{a}$ by summing up its elements. We aim at improving the performance of this kernel by exploiting the data level parallelism (DLP) of $\boldsymbol{a}$ using the SIMD capabilities available on the CPU. We will utilize the streaming SIMD extensions (SSE) to *manually* vectorize the kernel shown in Equation (1). We want to test our implementation for single precision data (32bit) and double precision data (64bit).

a) Implement vectorized code for the reduction kernel in Equation (1). A baseline version `gold_red` has already been implemented and can be used as a reference. You are asked to work on the items marked with "`TODO`" in the source file `vectorized_reduction.cpp` located in the directory with the same name inside the skeleton code directory. A guide with a possible workflow can be found in the `Readme.html` file within the source directory. You may use your browser to open the file. Furthermore, the Intel intrinsics guide[1] is a useful reference for this task.

b) In the previous subquestion we were concerned with the vectorization of our compute kernel in Equation (1), that is, the mapping of the execution flow to SIMD lanes for a single core. We can further exploit thread level parallelism (TLP) by mapping the SIMD execution flow to multiple cores using OpenMP. In this task you are going to extend the benchmarking routine in `vectorized_reduction.cpp` with TLP using the OpenMP framework. You can find further hints for this task in the `Readme.html` file and the comments in the code.

c) You can measure the performance of your vectorized code by running 'make measurement' on Euler. The job will create two `pdf` files, one for 32bit precision and another for 64bit precision results, each with speedup plots for a small n and another with a large n.

   i) What is the maximum speedup you expect for 32bit precision and 64bit precision data?

[1] https://software.intel.com/sites/landingpage/IntrinsicsGuide/

ii) Study the speedup plots in the generated `pdf` files and clearly explain the reason for differences you may observe for a small vector size n and a large vector size n (if there are any).

Question 2: Intel SPMD Program Compiler (ISPC) (42 points)

Manual vectorization can be cumbersome and requires the use of macros to easily switch between different floating point precision. ISPC is a compiler that helps the programmer to avoid such difficulties. As a result, you will be able to write optimized code faster, often with satisfying up to excellent results. This depends on the complexity of the code you want to optimize and yourself. It is very easy to write vectorized code that performs *worse* than the baseline version (true for manual optimizations as well as with ISPC). In this exercise we will utilize the single program multiple data (SPMD) programming model used by ISPC to map program instances to the SIMD lanes on the hardware. We will study ISPC together with the general matrix multiplication kernel (GEMM)

$$C = \alpha AB + \beta C, \tag{2}$$

where $A \in \mathbb{R}^{p \times r}$, $B \in \mathbb{R}^{r \times q}$ and $C \in \mathbb{R}^{p \times q}$ are matrices. For this exercise, we consider the scalars $\alpha = 1$ and $\beta = 0$.

ISPC is used to optimize a performance critical kernel (usually defined in a function, for example) and then compile optimized machine code for that kernel only. In order to use it in our main application code, we need to *link* to the optimized code at compile time. The application code for this exercise is contained in the file `gemm.cpp` inside the `ispc_gemm` directory in the skeleton codes folder. We want to target SSE2 and AVX2 instruction sets, which are both supported by the CPUs on the euler nodes. You are asked to complete the items marked with "`TODO`" in the skeleton codes. Have a look at the `Readme.html` file inside the source directory for a suggestion of how to solve the exercise and further tips. The ISPC documentation[2] is a helpful resource for this task. Your code should compile for 32bit precision and 64bit precision data (single precision and double precision).

a) Start with a baseline implementation of the kernel in Equation (2) in the application code `gemm.cpp`. You can compile and test your code with

```
make debug=true gemm_serial
```

You may omit the debug flag if you do not need debugging symbols in your code. Your baseline GEMM implementation should return a norm of truth of 255.966 for double precision data and 256.211 for single precision data.

b) To get started with the ISPC compiler, you need to install it. You can install it with

```
make install_ispc_linux_x86_64
```

This works on Euler or other 64bit Linux distributions. See also the `Readme.html` file. Windows and MacOS binaries can be downloaded here[3]. All explanations given in this exercise were tested with the Linux binary of ISPC.

c) Complete the ISPC related `Makefile` flags and implement the ISPC code for the kernel of Equation (2) in the file `gemm.ispc`. We target optimized kernels for SSE2 and AVX2 instruction sets. You can compile and link to ISPC code with the following (default) target

```
make debug=true gemm
```

[2] https://ispc.github.io/ispc.html
[3] https://ispc.github.io/downloads.html

You may omit the debug flag if you do not need debugging symbols in your code. You can submit a job on Euler to test your code using

```
make job
```

For convenience, you may want to work with an interactive node on Euler to omit the latency associated with submitting jobs to the queue.

d) What are the speedups you expect for your SSE2 and AVX2 optimized kernels? Report two numbers for each optimization, one for single precision data and one for double precision data. If your ISPC code does not reach these expectations, please state the reason for this behavior.

e) If there is a non-zero error associated with your optimized kernel, explain the reason for this error.

f) Do you observe differences in errors generated by the SSE2 and AVX2 instruction sets? If so, why is that?

Question 3: Implementing a distributed reduction (45 points)

In this question, you will use MPI to calculate the following sum:

$$x_{\text{tot}} = \sum_{n=1}^{N} n = 1 + 2 + 3 + \ldots + (N-1) + N \tag{3}$$

a) Fill in the missing part in the Makefile in order to compile the code with MPI support.

b) Validation of HPC code is an important subject. For example, there is an analytic formula for the above sum. Use this to check if your implementation is correct. To this end, implement the function `exact(N)`. **Hint:** A young C.F. Gauss found the formula in elementary school.

c) Initialize and finalize MPI by filling the corresponding gaps in the skeleton code.

d) Each rank performs only a part of the sum. Distribute the work load reasonably in order to guarantee load balancing. Each rank should calculate the subsum

$$\text{sum}_{\text{rank}} = N_{\text{start}} + (N_{\text{start}} + 1) + \ldots + N_{\text{end}}, \tag{4}$$

where N_{start} and N_{end} are the corresponding variables in the skeleton file.

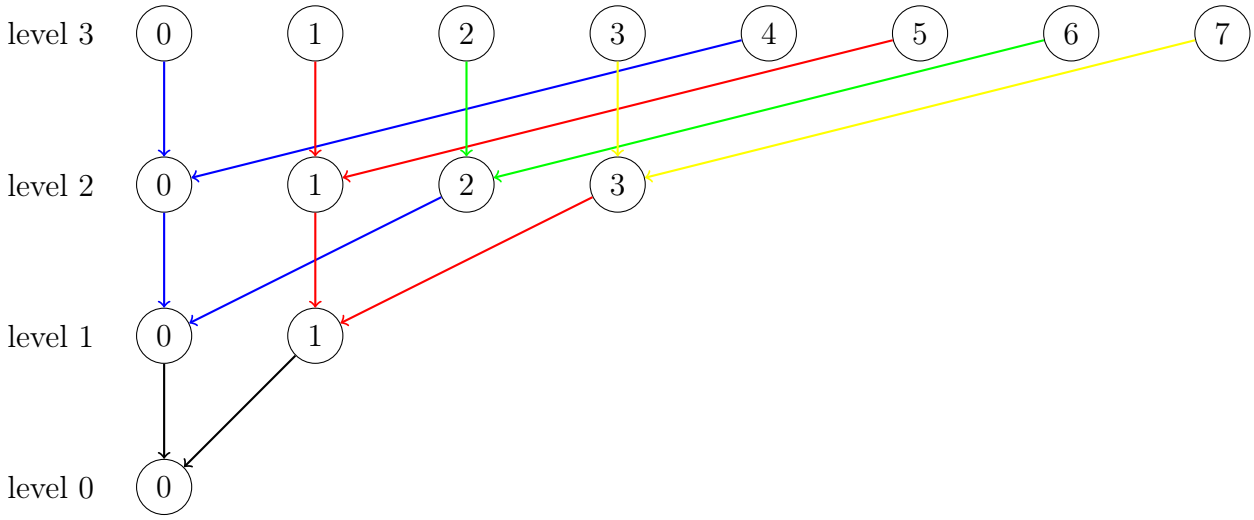

Figure 1: The communication pattern of a tree-like reduction. Each circle represents a rank, the number inside is the rank ID. Communication takes place along the arrows.

e) Finally, implement your own reduction. This can be done in a tree-like way as depicted in Fig. 1. Your task is to implement this scheme for the special case that the number of ranks is a power of 2, i. e.

$$|\text{ranks}| = 2^l, l \in \mathbb{N}_0 \tag{5}$$

f) What is the advantage of this scheme compared to the naive reduction? Name 2 advantages and quickly justify your answer. **Hint:** In the naive approach, every rank sends its elements directly to the master. The master then reduces all obtained elements by repeatedly applying the operation, in our case the sum.

Question 4: MPI Bug Hunt (16 points)

Find the bug(s) in the following MPI code snippets and find a way to fix the problem!

a)

```cpp
const int N = 10000;
double* result = new double[N];
// do a very computationally expensive calculation
// ...

// write the result to a file
std::ofstream file("result.txt");

for(int i = 0; i <= N; ++i){
    file << result[i] << std::endl;
}

delete[] result;
```

b)

```cpp
// only 2 ranks: 0, 1
double important_value;

// obtain the important value
// ...

// exchange the value
if(rank == 0)
    MPI_Send(&important_value, 1, MPI_DOUBLE, 1, 123, MPI_COMM_WORLD);
else
    MPI_Send(&important_value, 1, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD);

MPI_Recv(
    &important_value, 1, MPI_INT, MPI_ANY_SOURCE,
    MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE
);

// do other work
```

c) What is the output of the following program when run with 1 rank? What if there are 2 ranks? Will the program complete for any number of ranks?

```
MPI_Init(&argc, &argv);

int rank, size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int bval;
if (0 == rank)
{
    bval = rank;
    MPI_Bcast(&bval, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
else
{
    MPI_Status stat;
    MPI_Recv(&bval, 1, MPI_INT, 0, rank, MPI_COMM_WORLD, &stat);
}

cout << "[" << rank << "] " << bval << endl;

MPI_Finalize();
return 0;
```