Evaluating and optimising compiler code generation for NVIDIA Grace

Citation for published version:

Digital Object Identifier (DOI):
10.1145/3673038.3673104

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Publisher's PDF, also known as Version of record

Published In:
ICPP '24
Evaluating and optimising compiler code generation for NVIDIA Grace

Ricardo Jesus
rjj@ed.ac.uk
EPCC, The University of Edinburgh
Edinburgh, United Kingdom

Michèle Weiland
m.weiland@epcc.ed.ac.uk
EPCC, The University of Edinburgh
Edinburgh, United Kingdom

1 INTRODUCTION

Compilers are programs used to translate programs written in one programming language into programs written in another language, such as machine code. During this translation process usually a significant number of optimisations is attempted to ensure the resulting program executes as fast as possible on a given processor micro-architecture. Newer micro-architectures such as those from the AArch64 family, more and more prevalent in high-performance computing (HPC), are often at risk of missing out on such optimisations as compiler technology has had less time to mature to cater to their features than when compared to more traditional architectures. As a result, they might not achieve their full performance potential.

In this paper we explore the performance of the main optimising compiler toolchains available for high-performance AArch64 processors on the NVIDIA Grace CPU. Overall, the key contributions of our paper are:

- We evaluate the performance of the code generated by the ACFL, GNU, LLVM, and NVHPC compilers for the RAJA Performance Suite on the NVIDIA Grace CPU;
- We perform detailed root cause analyses for those kernels where LLVM exhibits the worst performance and provide a detailed explanation for each case;
- We recommend compiler flags or pragmas to solve some of the performance issues exhibited by LLVM;
- Where necessary, we implement compiler optimisations to improve LLVM performance on those kernels;
- And finally, we evaluate the optimisations (we achieve up to 70% performance improvement) and their general applicability to other codes and architectures.

The remainder of this paper is organised as follows. In Sec. 2 we provide the essential background needed for this paper, and in Sec. 3 we present the methodology we employ throughout. Then, in Sec. 4 we carry out a preliminary analysis of the performance of compilers on Grace. In Sec. 5 we take a deep dive into the main kernels where LLVM does worst and present possible optimisations. In Sec. 6 we evaluate those optimisations and discuss their applicability to other architectures and codes. In Sec. 7 we discuss related work and finally, in Sec. 8, we conclude the paper and point out directions for future work.

2 BACKGROUND

In this section we summarise the main background required for this paper. In particular, we present the main features of the NVIDIA Grace CPU, the RAJA Performance Suite, and the compilers used in this work.
2.1 The NVIDIA Grace CPU

The NVIDIA Grace CPU is a general-purpose processor developed by NVIDIA and based on the Neoverse V2 platform from Arm. It was designed to achieve performance leadership on HPC and artificial intelligence (AI)/machine learning (ML) workloads [2]. The CPU is used in a few NVIDIA systems including the NVIDIA Grace CPU Superchip, which features two Grace CPUs interconnected with NVIDIA NVLink Chip-to-Chip (C2C) technology, and the NVIDIA Grace Hopper Superchip, which combines a Grace CPU with a Hopper GPU (also interconnected with NVLink). In this work we utilise a Grace CPU in a Hopper-Hopper configuration. In Tab. 1 we list the main features of the processor. In sum, it features 72 Armv9 Neoverse V2 cores and is equipped with 480GB of LPDDR5X memory with error-correction code (ECC) with a memory bandwidth of 512GB/s. In terms of single-instruction multiple-data (SIMD) capabilities, each core is equipped with four arithmetic and logic unit (ALU) 128-bit SIMD pipelines supporting the Advanced SIMD (ASIMD), Scalable Vector Extension (SVE) and Scalable Vector Extension 2 (SVE2) extensions.

Table 1: Specifications of the Grace CPU.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISA</td>
<td>Armv9.0-A</td>
</tr>
<tr>
<td>Number of processor cores</td>
<td>72 Arm Neoverse V2 cores</td>
</tr>
<tr>
<td>SIMD units</td>
<td>4x128-bit SVE2</td>
</tr>
<tr>
<td>L1U/L1D cache size</td>
<td>64 KiB / 64 KiB</td>
</tr>
<tr>
<td>L2 cache size</td>
<td>1 MiB per core</td>
</tr>
<tr>
<td>L3 cache size</td>
<td>114 MiB</td>
</tr>
<tr>
<td>Peak FLOPS (double/single/half precision)</td>
<td>3.6 T/7.1 T/14.3 T</td>
</tr>
<tr>
<td>Memory bandwidth</td>
<td>512 GiB/s</td>
</tr>
<tr>
<td>Memory capacity</td>
<td>480 GiB LPDDR5X</td>
</tr>
</tbody>
</table>

2.2 The RAJA Performance Suite

The RAJA Performance Suite (RAJAPerf) is a benchmark suite designed to assess the performance of parallel programming models and runtimes on a wide range of loop-based kernels commonly found in HPC codes [5]. In total, the latest release of the suite (at the time of writing) contains 69 kernels written in C++14, and parallel libraries and runtimes such as OpenMP, CUDA, RAJA (which it was originally developed for), amongst others. The kernels it contains are mainly extracted from HPC applications and other benchmark suites. Each kernel is typically implemented in a number of "variants" that correspond to the library or runtime used (e.g. baseline sequential and OpenMP versions). To allow for a fair performance comparison between programming models, all variants of a kernel perform roughly the same mathematical operations and their loop bodies are implemented identically.

The kernels are partitioned into six main groups (Algorithm, Apps, Basic, Lcals, Polybench, and Stream), where the group name alludes to the origin and/or algorithmic patterns of its kernels. As an example, the "Apps" group contains a collection of kernels extracted from real-world HPC applications, whereas kernels in the "Lcals" group are extracted from the Livermore Compiler Analysis Loop Suite [4]. Each kernel is typically implemented in several variants that test a given kernel across different backends and implementation strategies. In total, there are 14 different such variants. In this work in focus on two, Base_Seq and Base_OpenMP, which correspond to baseline versions implemented using conventional sequential and OpenMP parallel loops, respectively. Besides wall-clock timing and other performance metrics, RAJAPerf computes and outputs checksums for all variants run. These checksums are used to ensure that all variants have run successfully.

2.3 AArch64 compilers for HPC

In this section we discuss the main AArch64 compilers considered in this work. We focus on compilers commonly used in an HPC setting. We do not discuss compilers such as the Cray or Fujitsu compilers, which are also able to target AArch64 hardware, as we did not have access to them for this work.

2.3.1 ACFL. The Arm Compiler for Linux (ACFL)² is a compiler toolchain developed by Arm Limited specifically designed for compiling code for Arm-based processors on a Linux operating system. The compiler suite consists of optimising C/C++ and Fortran compilers based on LLVM, and the Arm Performance Libraries (APL), a set of libraries providing optimised standard core math libraries for numerical applications, ensuring optimised performance across Arm-based architectures.

2.3.2 GNU. The GNU Compiler Collection³ is a suite of open-source compilers developed by the GNU Project offering support for various programming languages such as C, C++, Objective-C, Fortran, and others. The GNU compilers are widely used across diverse platforms and operating systems due to their robustness, portability, and extensive feature set. They provide advanced optimisation techniques to improve code performance along with support for numerous architectures, including AArch64.

2.3.3 LLVM. LLVM⁴ is an open-source compiler infrastructure toolchain known for its versatility and performance. LLVM operates as a collection of reusable libraries and tools designed to facilitate the construction of compilers and other programming language-related tools. One of its key features is its emphasis on intermediate representation (IR), a platform-independent representation akin to assembly code used within the compiler pipeline, enabling advanced optimisation techniques and cross-language interoperability. LLVM includes the Clang C/C++ compiler frontends and, as of recently, it features a Fortran frontend called Flang (the latter is an independent project from "Classic Flang"⁵, an open-source out-of-tree Fortran compiler based on pgfortran, a Fortran compiler from PGI/NVIDIA, that targets LLVM but is not part of the LLVM project in itself).

2.3.4 NVHPC. The NVHPC compilers⁶, part of the NVIDIA HPC SDK, are a suite of compilers developed by NVIDIA tailored for

1 https://github.com/LLNL/RAJAPerf/releases/tag/v2023.06.0
HPC applications on NVIDIA CPUs and GPUs. The suite consists of compilers for C, C++, and Fortran, with a focus on optimising code for parallel execution. Similarly to the ACFL, the NVHPC compilers are also based on LLVM.

3 METHODOLOGY

In this section we describe the methodology employed in the remainder of this paper. The compiler versions and optimisation flags used to build RAJAPerf are presented in Tab. 2. All tests were run in isolation and with exclusive use of the system. Furthermore, we utilise the OpenMP flags OMP_PLACES=cores OMP_PROC_BIND=true to pin threads to cores and use a total of 72 threads (the number of cores in a Grace chip) for parallel runs (OMP_NUM_THREADS=72). The CPU frequency governor is set to "performance" on all cores (scaling max frequency around 3.4 GHz).

Table 2: Compiler versions and optimisation flags used.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Version</th>
<th>Optimisation flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACFL</td>
<td>23.10</td>
<td>-O3 -ffast-math -mcpu=native -arch=lp -fopenmp</td>
</tr>
<tr>
<td>GNU</td>
<td>12.3</td>
<td>-O3 -ffast-math -mcpu=native -fopenmp</td>
</tr>
<tr>
<td>LLVM</td>
<td>18.1.2</td>
<td>-O3 -ffast-math -mcpu=native -fopenmp</td>
</tr>
<tr>
<td>NVHPC</td>
<td>24.3</td>
<td>-O4 -fast -mp -Munroll -Mvect</td>
</tr>
</tbody>
</table>

Furthermore, in the results discussed henceforth we often use relative performance scores to compare the performance of different compilers. These scores are obtained by normalising the runtime obtained by the code generated by each compiler on each kernel by the runtime obtained by the best compiler on the respective kernel. More precisely, if we let $T_c(k)$ represent the runtime of kernel $k$ compiled by compiler $c$ and $T^\text{best}(k)$ the best (smallest) observed runtime of kernel $k$ across all compilers, then we define $P_c(k)$, the relative performance of compiler $c$ on kernel $k$, as

$$P_c(k) = \frac{T^\text{best}(k)}{T_c(k)}.$$

Note that $0 < P_c(k) < 1$, i.e., $k$. The reason for using this metric is that (i) normalising all performance scores to the same range facilitates comparisons across compilers and kernels whose runtime could otherwise lie in very different scales, (ii) it makes the performance gap of a compiler on a given kernel compared to the best compiler more evident (for example, if a compiler achieves a relative score of 20% = 0.2 on a given kernel, then it was 0.2 times slower than the best compiler, and could in theory be improved at least by that much), and (iii) it allows us to summarise performance scores for multiple kernels into a single, meaningful number (e.g., as done in Fig. 1 to aggregate kernels by their group). Unless otherwise stated, all runtimes used to compute relative performance scores correspond to the median of 100 runs.

Finally, we note that NVHPC failed to compile the FIRST_MUN kernel with OpenMP, because the kernel utilises a user defined function to perform a reduction and this is currently unsupported by NVHPC. We have disabled building this kernel for NVHPC.

4 EVALUATING COMPILER PERFORMANCE

In Fig. 1 we show the relative performance each compiler obtains on the RAJAPerf kernel groups for Base_Seq (baseline "sequential") and Base_OpenMP (baseline threaded parallel) runs. Our intention is to give an overview of how the different compilers perform with respect to each other before delving into the causes for suboptimal performance in detail in the next sections. The relative performance score of each group was obtained by computing the geometric mean of the relative performance scores the compiler obtains on the kernels of the corresponding group. For completeness, we also plot the average performance each compiler obtains across all kernels as a fictitious group named "Overall".

Sequential runs. Performance of sequential runs is key to understanding the quality (i.e. efficiency) of the code generated by compilers. Significant differences in the performance of two compilers will arise if, for example, one compiler fails to vectorise a loop or picks a worse (less efficient) instruction mix to implement a given operation. Ultimately, these factors play a crucial role in both sequential and parallel performance, as in most cases parallelism is achieved by running the same program on multiple data. Therefore, all else being equal (e.g. performance of the parallel runtime and memory locality effects), single-thread performance governs the efficiency of the code.

Generally, all compilers we tested do reasonably well on sequential runs (Base_Seq), with the compiler that achieves the lowest average performance across all kernels still being within 86% of the performance of the best compiler. The group where the compilers are most even is Stream, which corresponds to kernels that test sustained memory bandwidth doing simple operations such as $A = B + AC$, and other such patterns. All the compilers achieve virtually the best performance in these kernels. As the kernels get more complex in groups Basic, Lcals, and Polybench, we see that the performance of the compilers tends to vary more widely but still remain comparable (gaps of less than 10%). Nevertheless, ACFL is consistently the compiler that generates the fastest code across these groups, achieving average performances of 97%, 98% and 96% (respectively). The GNU compiler in particular trails behind the others in the Lcals group, achieving an average performance of 82%. In the Algorithm group, which tests library functions such as memcpy, memset, sort and others, the fastest compilers are NVHPC (99%) and GNU (97%). The result of NVHPC underscores the efforts NVIDIA have placed into optimising these core library functions for the Grace GPU. Conversely, and somewhat surprisingly, in the most complex kernel group Apps, which contains more complex codes that mimic real HPC applications, NVHPC performs the worst amongst all others compilers by a significant margin, achieving an average performance of 68%. Coincidentally, this is also the case that shows largest deviation, indicating that this result is driven by a few kernels where NVHPC does very poorly (compared to the other compilers), and not a systematic problem in code generation.

Parallel runs. Overall, all but one compiler do well on parallel runs, with the three top performing compilers achieving average...
in the investigation. The remaining compilers, ACFL and NVHPC, slower than other compilers and those cases warrant further investigation. The remaining compilers, ACFL and NVHPC, exhibit similar trends to those seen in sequential runs.

performances (across all kernels) ranging from 82% (NVHPC) to 92% (LLVM) of best performance. The big outlier is GNU, which shows an average parallel performance of 60%, compared to the 89% it achieved on sequential runs. Furthermore, as evidenced by Fig. 1-b, the deviation of this result is relatively small across most kernel groups, which suggests that GNU is systematically slower than the remaining compilers. Given that GNU showed good performance in sequential runs—which indicates it generates efficient code—, this suggests that the culprit for the lower performance in parallel runs is the parallel runtime implementation being used, libgomp, which can be adding considerable overheads compared to the remaining compilers. This implies that improving parallel performance on GNU is not a matter of improving GNU itself, but rather improving its implementation of the OpenMP runtime. Indeed, in the Stream group, which as we have mentioned before contains mostly straightforward loops, GNU achieves an average performance of 99% on sequential runs (in other words, on par with all the other compilers), but only 44% on parallel runs, with very minimal deviations. This suggests the overheads of the runtime can be hampering GNU’s performance on parallel runs by almost 2x. Something similar happens in the Algorithm group and, to a lesser effect, in the remaining groups. Meanwhile, LLVM turned out to be the fastest compiler overall and across all groups, achieving over 90% performance in all cases (for parallel runs). Nevertheless, this does not mean that LLVM is the best compiler for all kernels—indeed as we will see in Sec. 5, in some cases LLVM can be more than 2x slower than other compilers and those cases warrant further investigation. The remaining compilers, ACFL and NVHPC, exhibit similar trends to those seen in sequential runs.

5 INVESTIGATING AND OPTIMISING LLVM CODE GENERATION
In this section we analyse in detail the 11 kernels where LLVM performs the worst on sequential runs, which we plot in Fig. 2. As the figure shows, on these kernels, the sequential relative performance of LLVM ranges from approximately 0.45 to around 0.82. Furthermore, a big portion of the kernels where LLVM exhibits poor sequential performance are kernels where it also exhibits poor parallel performance, and this effect would be even more prevalent if GNU did not exhibit the general performance problems on parallel runs we have seen in Sec. 4. Thus, from the viewpoint of optimisation, it is sensible to focus on sequential runs, understand the problems with code generation at that level and improve them as these improvements will generally percolate into the parallel runs as well. Similarly, it is worth noting that most of the kernels where LLVM does poorly are kernels where ACFL, NVHPC, or both do poorly too. This happens because both of these compilers utilise the LLVM infrastructure as mentioned in Sec. 2.3, and thus go through similar optimisations (except where the vendors have purposely optimised their compilers). Therefore, any improvements on LLVM will percolate into these other compiler toolchains and have a wide impact.

In the remainder of this section we take a closer look at each of the kernels in Fig. 2 and point out ways, either through compiler flags, pragmas, or compiler optimisations, whereby we can bring LLVM to deliver better performance (i.e. generate more efficient code), typically bringing it on par with the performance of each kernel’s best compiler. We use pseudo-code to help further illustrate the code patterns that exhibit performance issues.

Figure 1: Average relative performance each compiler obtains on each of the RAJAPerf kernel groups and overall across all kernels for (a) Base_Seq (sequential) and (b) Base_OpenMP (parallel) variants. Averages obtained by computing the geometric mean of the relative performance scores of the kernels of the corresponding group. Vertical error bars show the geometric standard deviation factor of the scores.
5.1 Interleaving

5.1.1 Algorithm_REDUCE_SUM and Basic_REDUCE3_INT. Algorithm_REDUCE_SUM and Basic_REDUCE3_INT are kernels that test code generation for loops that perform reductions. In particular, the former performs a sum reduction on an array of double-precision floating-point values, whereas the latter performs sum, min, and max reductions on an array of 32-bit integers. Below we illustrate the computations they carry out in C-like pseudocode:

```c
// Algorithm_REDUCE_SUM
double sum = 0.0;
for (long i = 0; i < m; ++i)
    sum += x[i];

// Basic_REDUCE3_INT
int vsum = 0;
int vmin = INT_MAX;
int vmax = INT_MIN;
for (long i = 0; i < m; ++i) {
    vsum += vec[i];
    vmin = vec[i] < vmin ? vec[i] : vmin;
    vmax = vec[i] > vmax ? vec[i] : vmax;
}
```

The two best performing compilers for Algorithm_REDUCE_SUM, GNU and NVHPC, implement the reduction by vectorising the loop and interleaving it by four, thereby accumulating the partial sums in four vector registers. Then, once all elements have been accumulated, the partial sums are reduced to the final result in a tree fashion by a series of other “add” instructions. LLVM follows a very similar approach, with the key difference that it only interleaves the loop by two. Given the Neoverse V2 has four 128-bit vector arithmetic (and logic) units—and thus can potentially execute up to four fadd instructions simultaneously—it is beneficial from a performance point of view to interleave the loop by four as done by GNU and NVHPC. In LLVM, this can be achieved either globally with a compiler flag, -mllvm -force-vector-interleave=N, or on a loop by loop basis with the pragma:

```c
#pragma clang loop interleave_count(N)
```

where N is a positive integer. Using the latter approach (with N = 4), we can get LLVM to interleave the loop by four as the two previously mentioned compilers, thus generating more efficient assembly where four vector registers are utilised to accumulate the partial sums. The performance issues exhibited by Basic_REDUCE3_INT are very similar—LLVM also vectorises the kernel successfully but only interleaves it by two, whilst GNU interleaves it by four and thus achieves significantly better performance. Using a similar strategy with pragmas enables LLVM to obtain significantly better performance, comparable to that of GNU.

5.2 Vectorisation

5.2.1 Algorithm_MEMCPY. As its name suggests, the Algorithm_MEMCPY kernel implements a straightforward memory copy operation as exemplified below:

```c
for (long i = 0; i < m; ++i)
    y[i] = x[i];
```
NVHPC is the best compiler for this code, with the other compilers achieving approximately 80% of its efficiency on sequential and parallel runs (with the exception of GNU, which only scores 37% on parallel runs). NVHPC compiles the kernel into a call to a function of libnvc.so (automatically linked by the nvc++ compiler) where the actual copying of the data occurs. The hot region of this function is a loop with four ld1d/stp instruction pairs that operate on two 128-bit registers, plus a few other instructions for the loop’s bookkeeping. LLVM’s implementation is not very different, consisting of two (single-register) ld1d/st1d instruction pairs plus a few bookkeeping instructions7. The crucial difference lies in that the code emitted by the compilers handles a different number of elements per iteration. On NVHPC, a total of 16 64-bit elements are copied per iteration (four per ld1d/stp pair), whereas LLVM only copies 4 such elements per iteration (two per ld1d/st1d pair). Furthermore, though NVHPC handles four times more elements per iteration than LLVM, it does so using just short of two times the number of instructions. The better ratio of “number of instructions to work done” that NVHPC achieves is made possible by the usage of the ld1d/stp instructions, which load/store pairs of 128-bit vector registers at a time (when used with Q registers). The ld1d/st1d instructions, on the other hand, only load/store a single vector register at a time, which on the Neoverse V2 micro-architecture both ld1d (st1d) and ld1d (st1d) instructions have the same latency of 6 (2) cycles when the memory access hits L1 data cache, the ld1d (st1d) instructions are effectively handling twice as much data as the ld1d/st1d pairs, and whilst this does not necessarily mean the former set of instructions is twice as fast as the latter (since performance will be limited by other factors such as memory bandwidth and resource availability), it does mean that the CPU will have to dispatch a smaller number of instructions to perform the same amount of work, and make better use of its resources.

In practice, we can get LLVM to emit ld1d/stp instructions by disabling scalable vectorisation (which is relying on to emit ld1d/st1d instructions).-mlvm -scalable-vectorization=off or on a loop by loop basis with a pragma,

```c
#pragma clang loop vectorize_width(N[, fixed|scalable])
```

where N is a positive integer and the second (optional) parameter controls the type of vectorisation requested and defaults to fixed. To test this change we have chosen the latter option and marked the kernel’s loop with the pragma below:

```c
#pragma clang loop vectorize_width(8)
```

This is functionally equivalent to specifying fixed-width vectorisation (to obtain the ld1d/stp instructions) and interleaving the loop enough to obtain four such instruction pairs, leading to a new instruction mix that processes the same number of elements per iteration as NVHPC and thus achieves comparable performance.

5.2.2 Apps_EDGE3D. Apps_EDGE3D is a larger code that implements a differential equation solver using the finite element method (FEM). Overall, LLVM exhibits approximately a 20% slowdown in this kernel compared to GNU. This performance gap results from the compiler’s implementation of a single function, inner_product, which computes the inner product of two 3-dimensional basis vectors as illustrated below in C-like pseudocode:

```c
for (int p = 0; p < P; p++) {
   const double txi = basis_2_x[p];
   const double tji = basis_2_y[p];
   const double tzi = basis_2_z[p];

   const int m0 = (is_symmetric && (M == P)) ? p : 0;
   for (int m = m0; m < M; m++) {
      const double txj = basis_1_x[m];
      const double tij = basis_1_y[m];
      const double tzi = basis_1_z[m];

      matrix[p][m] = weight*txi*txj + tij + tzi;
   }
   matrix[m][p] = matrix[p][m];
}
```

Importantly, as demonstrated in line 15 in the code above, to speedup the computation the benchmark checks if the problem is symmetric and if so avoids computing the products of approximately half of the elements of matrix, resorting to copying them instead. It turns out this algorithmic optimisation prevents LLVM from vectorising the code automatically as it assumes the copy in line 16 is an unsafe memory dependency as reported when compiling with loop vectorisation remarks (-pass-analysis=loop-vectorize):

```c
remark: loop not vectorized: unsafe dependent memory operations in loop.
use #pragma clang loop distribute(enable) to allow loop distribution to attempt to isolate the offending operations into a separate loop
```

GNU also identifies the same memory dependency, but versions the loop for vectorisation (i.e. generates a loop version without memory aliases) and at runtime checks whether the memory accesses alias or not, based on which it decides whether the optimised version can be used.

LLVM should arguably be able to follow a similar approach, though the same effect can be achieved by splitting the inner loop (in m) into two, moving the copying of data into a separate loop. This is indeed what is hinted at in the remark above with #pragma clang loop distribute(enable), but even when explicitly asked to do so, LLVM still fails to perform the transformation on its own. Nevertheless, by making this change manually, LLVM finally manages to vectorise the inner loop, leading to significantly more efficient assembly, comparable to GNU’s.

5.2.3 Apps_MASS3DPA. This is a kernel extracted from MFEM8 and CEED9 algorithms and implements the action of a 3D mass matrix via partial assembly. The code is organised in a series of loops of the form:

```c
for (int dy = 0; dy < MPA_D1D; dy++) {
   for (int dx = 0; dx < MPA_D1D; dx++) {
      MASS3DPA_N
   }
}
```

where MASS3DPA_N is a macro containing other loops, such as:

---
7We note that SVE2.1 supports versions of certain load/store instructions including ld1d/st1d that operate on more than one register. These are, however, not supported on the Grace CPU, which only supports SVE2.


There are six such loops in the code, and although their particular bounds and the exact operations they perform vary, they all follow roughly this structure.

The best compiler for this code is GNU, with LLVM being approximately 35% slower. When building the code with optimisation reports enabled, we see that both compilers fully unroll the loops contained in the macros and otherwise perform mostly the same optimisations. The key difference lies in that GNU chooses to automatically vectorise almost all of the loops around the macros (i.e. those in dx above), whereas LLVM does not. This is crucial for performance as all these macros contain reductions and copies in dx as those portrayed in lines 6 and 8 of the macro (respectively). We can bring LLVM to vectorise these loops by marking them with a pragma such as clang loop vectorize_width(4). By doing so for all the loops with these operations (including those which GNU does not vectorise by default), LLVM is able to achieve even better performance than GNU.

5.3 Loop invariant code motion

5.3.1 Basic_INIT_VIEW1D, Basic_INIT_VIEW1D_OFFSET, and Basic_NESTED_INIT. These kernels initialise an array according to a given expression. We group their discussion together since, as we will see below, the reason behind their poor performance is the same. Below we illustrated the computations each of them carries out in C-like pseudocode:

```c
1 // Basic_INIT_VIEW1D
2 for (long i = 0; i < m; ++i)
3 a[i] = (i+1) * v;
4
5 // Basic_INIT_VIEW1D_OFFSET
6 for (long i = 1; i < m+1; ++i)
7 a[i] = i * v;
8
9 // Basic_NESTED_INIT
10 for (long k = 0; k < nk; ++k)
11 for (long j = 0; j < nj; ++j)
12 for (long i = 0; i < ni; ++i)
13 a[i+ni*(j+nj*k)] = 0.00000001 * i * j * k;
```

Importantly, as the code snippet shows, all expressions in the loop bodies utilise the loops’ induction variables to compute the values to initialise. As we shall see, this is critical for performance in LLVM.

No single compiler is the best across all these three kernels. In particular, ACFL is best on Basic_INIT_VIEW1D and Basic_INIT_VIEW1D_OFFSET (on par in the former with NVHPC), and GNU generates the most efficient code on Basic_NESTED_INIT. As these kernels share the same major performance problem on LLVM, the rest of the discussion is going to focus solely on Basic_INIT_VIEW1D. Both ACFL and LLVM vectorise this kernel and interleave it by two, and in fact generate very similar assembly for it. Yet, ACFL is almost 70% faster than LLVM. In brief, both compilers start by forming two vector registers with the vector expansions of i+1, convert the values from integer to floating-point representation, and multiply the results by v to obtain (i+1)*v. The results are then stored to the array a. The key difference between the two compilers lies in the way they handle the update of the vector register containing the vector expansion of the loop’s induction variable i from one iteration to the next, and the implications this has on loop carried dependencies and instruction pipelining. In the case of ACFL, the register that holds the vector expansion of i is updated by a single increment by four, the number of elements the loop processes at each iteration, in a single instruction:

```c
1 add z4.d, z2.d, z2.d
```

Meanwhile, though functionally equivalent, LLVM utilises three instructions to achieve the same effect:

```c
1 add z4.d, z3.d, z2.d
2 add z5.d, z4.d, z2.d
3 mov z3.d, z5.d
```

In particular, instead of incrementing the register it uses to keep the vector expansion of i directly by four, it chooses to split the increment into two increments of two (the first two add instructions), and finally move the result to set up the next iteration (the last mov instruction). In practice, as the add and mov instructions on the Neoverse V2 have a latency of 2 cycles, this means that in the best case scenario the ACFL version of the code can potentially start executing a new iteration of the loop every two cycles, whereas the LLVM version can only start a new iteration every six cycles—which is highly detrimental for performance as it prevents the CPU from pipelining more instructions.

The behaviour exhibited by LLVM is a remnant of its vectorisation pass, which, because the loop is interleaved by two, introduces the two aforementioned chained increments. This can be seen in the snippet of LLVM IR below, where %vec.ind.next, which holds the update of the vector expansion of the induction variable, is computed through %step.add, which in turn is computed from %vec.ind (the actual vector expansion at the start of each iteration):

```c
1 %vec.ind = phi vscale x 2 x i64 [ %5, %vector.ph ], [ %vec.ind.next, %vector.body ]
2 %step.add = add vscale x 2 x i64 %vec.ind, %splat
3 %vec.ind.next = add vscale x 2 x i64 %step.add, %splat
```

As no source-level transformation could address this issue, we developed an LLVM peephole optimisation that improves loop invariant code motion (LICM) by looking for chains of binary operations such as those above and rewriting them to (i) associate operations whose operands are loop invariant together and (ii) move the resulting sub-expressions out of the loop. As an example, the LLVM IR above gets rewritten such that the intermediate increment is hoisted out of the loop and thus the vector index is updated in a single operation as illustrated below:

```c
1 %invariant.op = add vscale x 2 x i64 %splat, %splat
2 %vec.ind = phi vscale x 2 x i64 [ %5, %vector.ph ], [ %vec.ind.next, %vector.body ]
3 %vec.ind.next = add vscale x 2 x i64 %vec.ind, %invariant.op
```

Evaluating and optimising compiler code generation for NVIDIA Grace ICPP ’24, August 12–15, 2024, Gotland, Sweden
This optimisation enables others to kick in and leads to very efficient assembly, identical in performance to that of ACFL. Furthermore, though here we have only discussed it in the light of the Basic_INIT_VIEW1D kernel, as we have mentioned previously the other two kernels mentioned in this section, Basic_INIT_VIEW1D_OFFSET and Basic_NESTED_INIT, suffer from the same problem and thus this optimisation also improves their performance. Due to its general applicability, we have contributed it back into LLVM where it is currently under review.10

5.4 Other

5.4.1 Apps_FIR. This kernel computes the finite impulse response between a signal and an array of coefficients as exemplified below:

```c
#include <Arduino.h>

void update_coeffs(float *coeff, int coefflen)
{
    double sum = 0.0;
    for (int k = 0; k < coefflen; ++k)
    {
        sum += coeff[k];
    }
    *coeff = sum / coefflen;
}
```

The key performance problem with the kernel lies in the bounds of its inner loop (in j, where the impulse response is computed). The loop runs through the elements of the coefficients array coeff at indices ranging from 0 to coefflen. At runtime, coefflen takes the value of 16 (the number of elements of coeff), and thus, in practice, the loop runs through all the elements of coeff. However, due to the way coefflen gets initialised in the benchmark, this is not obvious to the compiler. In fact, coefflen gets initialised indirectly through an attribute of a supporting object which itself gets initialised (in its class’ constructor) with the value of 16 and is henceforth never modified.

The optimisation described in Sec. 5.2.3, suffers from the same problem. In practice, the loop bounds D1D and Q1D are small constants (4 and 5, respectively).

None of the compilers we tested were able to assert the value of coefflen at compile time as this would involve nontrivial pointer analysis. However, the best compiler for this code (GNU) was still able to generate very efficient code by unrolling the inner loop (in j) by the length of the coeff array, 16, and placing runtime checks as necessary to ensure coefflen is never overrun. This enabled it to carry out a series of other optimisations such as fully vectorising the inner loop, replacing some of the products and additions by negations and subtractions, and hoisting the loads of elements of coeff to avoid repeatedly (re)loading them from memory. LLVM also vectorises and interleaves the loop to process 4 elements per iteration, but does not carry out these extra optimisations and therefore only achieves about half the performance of GNU.

To verify whether LLVM might (hypothetically) be able to carry out the optimisations demonstrated by GNU in the case where it could determine the length of the coefficients array at compile time, we explicitly initialised coefflen to 16. In this case, LLVM automatically unrolls the inner loop fully but, instead of vectorising it, it vectorises the outer loop (in i) instead. In doing so, it also carries out efficient arithmetic simplifications like those exhibit by GNU (e.g. replacing some of the products and additions by subtractions) and removes the loads of coeff, thus generating very efficient assembly. Notwithstanding the fact that explicitly initialising coefflen to 16 makes the compiler’s job much easier, one can argue that in most real codes such variables would likely not be initialised through so many indirections. Thus, in most comparable real world cases LLVM is in fact capable of generating efficient code for such code patterns, and the inefficiencies shown in RAJAPerf is primarily a product of the construction of the benchmark itself.

5.4.2 Apps_MASS3DEA. Similarly to Apps_MASS3DPA discussed in Sec. 5.2.3, Apps_MASS3DEA is another kernel extracted from MFEM and CEED algorithms. However, whilst the computations in the former were carried out in several independent loops, the main computations in this kernel are carried out in a chain of several nested loops. With non-crucial parts of the code (from the point of view of performance) omitted, the computation resembles the following:

```c
for (int i = 0; i < NE; ++i) {
    // ...
}
```

In practice, the loop bounds D1D and Q1D are small constants (4 and 5, respectively).

This is the kernel where LLVM achieves the lowest relative performance amongst all kernels, being more than two times slower than the best compiler GNU. When we generate optimisation reports from both compilers, we see that they both fully unroll the two most inner loops in k2 and k3. However, whilst then LLVM tries to partially unroll the loop in k1, GNU proceeds to vectorise the “outer loop” in j3. Vectorising the loop in j3 whilst leaving the loop in k1 untouched turns out to be highly efficient as it allows GNU to vectorise the kernel whilst keeping register pressure (and spilling) to a minimum. LLVM, however, is unable to carry out this optimisation and vectorise the loop in j3 without also unrolling the loop in k1, neither with its default loop vectoriser11 nor with VPlan12, an alternative vectorisation strategy that supports outer loop vectorisation and which can be enabled with -mlvm-enable-vplan-native-path=true. This is inefficient as, even though the constants of the loop bounds are relatively small, fully unrolling the three inner loops in k1 to k3 and vectorising the loop in j3 leads to too high register spilling, whereby the compiler as to keep utilising stack memory to store temporary values.

Table 3: Runtimes and speedups obtained with the optimisations described in Sec. 5. Green bars show the speedup obtained with LLVM, whilst orange bars show the performance gap that remains (if any) compared to the best compiler. White vertical gaps in green bars mark the performance of the best compiler prior to our optimisations (thus, to the right-hand side of white vertical bars we have the improvement of our optimisations over the previously best compiler). Median and median absolute deviation (MAD) of 100 runs.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>(Section discussed)</th>
<th>Variant</th>
<th>Original time (s)</th>
<th>Improved time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Median</td>
<td>MAD</td>
<td>Median</td>
<td>MAD</td>
</tr>
<tr>
<td>Algorithm_MEMCPY</td>
<td>(§5.2)</td>
<td>Base_Seq</td>
<td>0.02978</td>
<td>6.68e-04</td>
<td>0.02370</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base_OpenMP</td>
<td>0.00056</td>
<td>6.00e-06</td>
<td>0.00049</td>
</tr>
<tr>
<td>Algorithm_REDUCE_SUM</td>
<td>(§3.1)</td>
<td>Base_Seq</td>
<td>0.01025</td>
<td>3.71e-04</td>
<td>0.00817</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base_OpenMP</td>
<td>0.00403</td>
<td>5.00e-06</td>
<td>0.00040</td>
</tr>
<tr>
<td>Apps_EDGE3D</td>
<td>(§5.2)</td>
<td>Base_Seq</td>
<td>16.19034</td>
<td>4.03e-03</td>
<td>12.98819</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base_OpenMP</td>
<td>0.22790</td>
<td>1.82e-04</td>
<td>0.18891</td>
</tr>
<tr>
<td>Apps_REDUCE3_INT</td>
<td>(§5.1)</td>
<td>Base_Seq</td>
<td>0.00428</td>
<td>1.04e-04</td>
<td>0.00355</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base_OpenMP</td>
<td>0.00036</td>
<td>4.00e-06</td>
<td>0.00035</td>
</tr>
<tr>
<td>Apps_MASS3DEA</td>
<td>(§3.4)</td>
<td>Base_Seq</td>
<td>0.02719</td>
<td>2.00e-05</td>
<td>0.01516</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base_OpenMP</td>
<td>0.00045</td>
<td>1.00e-06</td>
<td>0.00026</td>
</tr>
<tr>
<td>Apps_MASS3DPA</td>
<td>(§3.2)</td>
<td>Base_Seq</td>
<td>0.14794</td>
<td>2.66e-04</td>
<td>0.08522</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base_OpenMP</td>
<td>0.00217</td>
<td>6.00e-06</td>
<td>0.00161</td>
</tr>
<tr>
<td>Basic_INIT_VIEW1D</td>
<td>(§5.3)</td>
<td>Base_Seq</td>
<td>1.33035</td>
<td>9.45e-04</td>
<td>0.77461</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base_OpenMP</td>
<td>0.02711</td>
<td>2.20e-04</td>
<td>0.01908</td>
</tr>
<tr>
<td>Basic_INIT_VIEW1D OFFSET</td>
<td>(§3.3)</td>
<td>Base_Seq</td>
<td>0.83590</td>
<td>3.31e-03</td>
<td>0.58005</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base_OpenMP</td>
<td>0.02704</td>
<td>2.12e-04</td>
<td>0.01911</td>
</tr>
<tr>
<td>Basic_NESTED_INIT</td>
<td>(§5.3)</td>
<td>Base_Seq</td>
<td>0.33037</td>
<td>2.59e-04</td>
<td>0.25245</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base_OpenMP</td>
<td>0.00989</td>
<td>9.20e-05</td>
<td>0.00827</td>
</tr>
<tr>
<td>Basic_REDUCE3_INT</td>
<td>(§5.1)</td>
<td>Base_Seq</td>
<td>0.00428</td>
<td>1.04e-04</td>
<td>0.00355</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base_OpenMP</td>
<td>0.00036</td>
<td>4.00e-06</td>
<td>0.00035</td>
</tr>
<tr>
<td>Lcals_PLANCKIAN</td>
<td>(§3.4)</td>
<td>Base_Seq</td>
<td>0.16809</td>
<td>3.06e-04</td>
<td>0.16817</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base_OpenMP</td>
<td>0.00262</td>
<td>1.30e-05</td>
<td>0.00262</td>
</tr>
</tbody>
</table>

Given this present limitation of LLVM, the most efficient version of this kernel we could obtain was achieved by fully and manually unrolling the three inner loops (k1 to k3), and leaving the loop in j3 untouched. Whilst this does not achieve the full performance of GNU’s version, it still provides a very significant uplift and places LLVM within 20% of GNU’s performance.

5.4.3 Lcals_PLANCKIAN. The Lcals_PLANCKIAN kernel is based on Planck’s law and performs the computation outlined below:

```c
1 for (long i = 0; i < m; ++i) {
2     y[i] = u[i] / v[i];
3     w[i] = x[i] / (exp(y[i]) - 1.0);
4 }
```

The fastest compiler on this kernel is ACFL, which generates code approximately 40% faster than the other compilers. The reason for this performance gap lies in that ACFL manages to automatically vectorise the whole loop, including the exponential (`exp`) function call. This is not the case with LLVM or GNU—the former vectorises part of the loop but serialises the calls to exp, and the latter does not vectorise the loop at all. ACFL achieves this by leveraging libamath, a library contained in ACFL and ArmPL which contains AArch64-optimised scalar and vector implementations of the standard C mathematical functions of `libm`. The routines in `libamath` are open-source and part of the Arm Optimized Routines13. We note that LLVM’s loop vectoriser should be able to automatically vectorise some intrinsic mathematical functions including `exp`, but in practice failed to do so in all our tests. Finally, even though NVHPC was able to automatically vectorise the loop, including the exponential calls, its performance on this kernel was the lowest across all compilers. This is likely because the code it generates contains inefficiencies such as excessive loads/stores which we do not observe on ACFL or the other compilers.

6 EVALUATING LLVM OPTIMISATIONS

In this section we evaluate the LLVM optimisations described in Sec. 5. To this end, in Tab. 3 we present the runtimes and corresponding speedups obtained with the aforementioned optimisations on

---

the kernels examined (where LLVM achieved the lowest performance), for sequential and parallel runs. As the table shows, our optimisations were overall successful in bridging the gap between the performance of LLVM and the best compilers, and often make LLVM the best compiler for kernels where it was previously performing poorly. In all but two kernels, our optimisations either place LLVM within a few percent of the performance of the best compilers, or indeed make LLVM surpass their performance. The two exceptions are the kernels Apps\_MASS3DEA and Lcals\_PLANCKIAN. As discussed in Sec. 5, these are kernels where we could not carry out the optimisations other compilers perform—in the case of the former because LLVM fails to perform outer-loop vectorisation, without which the loop in question cannot be vectorised without introducing high register spilling, and in the latter because LLVM fails to vectorise calls to the exp (exponential) math library function. Nevertheless, for Apps\_MASS3DEA some performance improvements can still be gained by other means, namely by fully unrolling the loop around which the vectoriser should perform outer-loop vectorisation. Furthermore, as we pointed out in Secs. 4 and 5, though our analyses focused primarily on sequential runs, we see similar performance improvements across sequential and parallel runs. This shows that improving general code generation indeed pays dividends in a wider scope—including when parallelising the code.

Moreover, we can see that no single optimisation technique (such as interleaving or vectorisation) is necessarily more important than or dominates the others—all of them are required, with approximately the same expression in the set of kernels we analysed, to achieve high performance. Similarly, we see that whilst some optimisations are "generally applicable" to any architecture, others are more geared towards the CPU’s architecture (AArch64) or even its micro-architecture (Neoverse V2). For example, the loop invariant code motion optimisation described in Sec. 5.3 for kernels such as Basic\_INIT\_VIEW1D and Basic\_NESTED\_INIT will generally benefit any architecture on loops that perform arithmetic operations with the induction variable, and the vectorisation of the loops of Apps\_MASS3DPA should benefit any architecture that supports vector extensions. Meanwhile, the decision of vectorising loops like those in Alg1\_thm\_MEMCPY with fixed-width or scalable vectors is something that generally affects AArch64 and other architectures with vector length specific and vector length agnostic instruction sets. Finally, deciding on values like optimal interleaving factors, as seen in kernels such as Alg1\_thm\_REDUCE\_SUM, is something that ultimately boils down to the micro-architecture and its characteristics such as number of vector units and dispatch width, for example.

7 \ RELATED WORK

The main features of the NVIDIA Grace CPU have been presented in [3], where the authors also provide some of the history behind its development. Despite being relatively recent hardware, a few works have analysed the performance of the Grace CPU for HPC usage. In [6] the authors evaluate the performance of the Grace CPU on HPC applications and benchmarks such as Gromacs, Open-FOAM, and HPCG, and compare its performance against that of several CPUs including Intel Sapphire Rapids, AMD Milan, and Fujitsu A64FX. Overall they find similar per-core performance compared to the Intel and AMD CPUs, with slightly better performance per-node. Importantly, besides focusing on different workloads than our paper, they do not analyse in-depth the performance of the codes they evaluate. Similarly, in [1] the authors compare the scalability performance of applications such as OpenFOAM, NEMO, and LAMMPS on NVIDIA Grace and Grace Hopper Superchips against the performance of the MareNostrum 4 supercomputer (equipped with Intel Skylake CPUs), and report speedups between 1.5 and 4.3x. Once again, besides considering fundamentally different workloads, the authors do not examine the codes in detail nor propose code generation optimisations to improve performance.

8 \ CONCLUSIONS AND FUTURE WORK

In this paper we have shown that the quality of code generated for the Grace CPU overall (i.e. the performance achieved) across compilers is reasonably good, even though the hardware has not been available long. Nevertheless, there is no single "best" compiler across all kernels. Using a benchmark suite such as RAJAPerf is useful to drill down into details to understand where performance might be suboptimal, and why. We showed how to improve performance for LLVM specifically, either through compiler flags and pragmas, or by implementing optimisations directly in the compiler. The LLVM ecosystem in particular lends itself well to this task, letting us explore and prototype optimisations quickly and efficiently. The optimisations we devised have enabled us to obtain speedups of over 70% in some kernels. Finally, although the optimisation work presented here has focused on LLVM, most of the optimisations we have suggested can be readily applied to the other compilers. In fact, the optimisations will likely also benefit languages other than C/C++ (e.g. Fortran) when compiled using the LLVM toolchain. Likewise, although some are AArch64 specific, most optimisations can benefit other architectures as well. In the future we plan to leverage the optimisation opportunities we have developed to enable LLVM to provide best performance for Grace out-of-the-box. Furthermore, we plan to investigate the performance issues exhibited by GNU on parallel runs.

REFERENCES


