EPJ Web of Conferences **245**, 01006 (2020) https://doi.org/10.1051/epjconf/202024501006
CHEP 2019

Compass SPMD: a SPMD vectorized tracking algorithm

Placido Fernandez Declara[1,2,*] and *J. Daniel* Garcia[2] on behalf the RTA LHCb project.

[1]EP-LBC, CERN, 1211–Geneve 23, Switzerland
[2]Department of Computer Science and Engineering, University Carlos III of Madrid, Madrid, Spain

Abstract. Compass is a SPMD (Single Program Multiple Data) tracking algorithm for the upcoming LHCb upgrade in 2021. 40 Tb/s need to be processed in real-time to select events. Alternative frameworks, algorithms and architectures are being tested to cope with the deluge of data. Allen is a research and development project aiming to run the full HLT1 (High Level Trigger) on GPUs (Graphics Processing Units). Allen's architecture focuses on data-oriented layout and algorithms to better exploit parallel architectures. GPUs already proved to exploit the framework efficiently with the algorithms developed for Allen, implemented and optimized for GPU architectures. We explore opportunities for the SIMD (Single Instruction Multiple Data) paradigm in CPUs through the Compass algorithm. We use the Intel SPMD Program Compiler (ISPC) to achieve good readability, maintainability and performance writing "GPU-like" source code, preserving the main design of the algorithm.

1 Introduction

LHCb is one of the large four experiments in the LHC at CERN. From 2019 it started an upgrade of its components and software for the physics data-taking period that will start in 2021. Its software will need to compute a collision rate of 30 MHz which generates a data throughput of 40 Tb/s that needs to processed in real-time [1]. As a novelty its event filter farm will be powered solely by general purpose computing resources: a software trigger. The software trigger source code is being updated to cope with the increased throughput rate, as an increased compute power is needed to better exploit the hardware. As part of it, various hardware alternatives are being considered, with multi- and many-core CPUs, co-processors and accelerators included.

2 UT tracking

The *Compass* algorithm computes the tracking for the UT (Upstream Tracker) sub-detector. UT tracking receives input data from the VELO (Vertex Locator) as reconstructed tracks, and hit information from the UT. The UT is located between the VELO and SciFi tracking subdetectors as depicted in Figure 1, which shows the tracking subdetectors and associated track types.

It is composed by four planes, where each plane is a single sided silicon strip detector. Each UT plane is composed of micro-strip sensors arranged in vertical staves. A UT plane

*e-mail: placido.fernandez@cern.ch

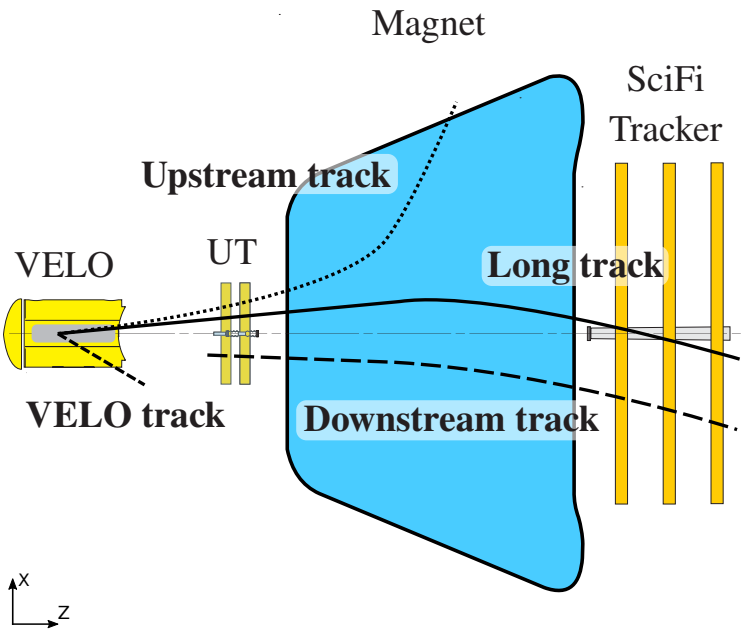

Figure 1. UT tracking subdetectors and tracks

can be divided into three regions with different geometry, where the inner-most region has a finer granularity. Tracks that traverse the UT can be slightly bent by the magnetic field, which poses a computing multiplicity problem to match VELO tracks to UT hit candidates for every event, as shown in Figure 2.

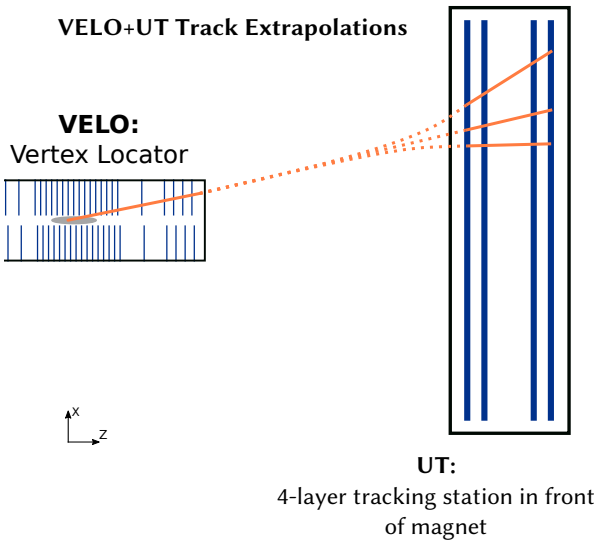

Figure 2. UT tracks extrapolation

```
typedef float<3> float3;

export void chi2_ispc (
  uniform const size_t N, uniform float chi2 [],
  uniform float3 x [], uniform float3 y [],
  uniform const float m, uniform const float q) {
    foreach(i = 0 ... N) {
      varying float3 expected_y = m * x[i] + q;
      chi2[i] = (y[i].x - expected_y.x) + (y[i].x - expected_y.x) +
                (y[i].y - expected_y.y) + (y[i].y - expected_y.y) +
                (y[i].z - expected_y.z) + (y[i].z - expected_y.z);
    }
}
```

Listing .1: ISPC source code sample

UT serves various purposes in the LHCb experiment, being the main ones: reconstructing charged particle trajectory that decay after the VELO sub-detector, reconstructing low momentum particles that are bent by the magnetic field and go out of acceptance before reaching the SciFi tracker, providing additional hit information used in conjunction with VELO and SciFi sub-detectors to reject tracks, providing momentum resolution for charged particles and decreasing the time needed to extrapolate VELO tracks to SciFi tracker by at least a factor three.

3 SPMD and ISPC

Allen's framework design allows the chain of algorithms that computes the full HLT1 to compile for both GPU and CPU architectures [2]. Allen is designed and optimized for GPUs; compilation for CPUs in Allen supports basic multithreading and does not vectorize the algorithms. The same target source code in Allen is able to exploit all threads in a GPUs through its *warps*[1] in an efficient manner, but when compiled for CPUs these won't map to vector lanes and the vectorization units will be underutilized. Explicit support in the form of vector instructions, intermediate libraries or directives is needed to exploit these resources.

The Intel SPMD Program Compiler (ISPC) [3] uses a variant of the C language to write sequential-like algorithms, but its execution model executes various *program instances* that run in parallel through the vector lanes. It presents an alternative way to exploit vectorization units by writing algorithms in a similar way as for GPUs. When implementing an ISPC program the variables that will run with different values across the vector lanes are explicitly indicated through the keywords *uniform* and *variant*, allowing the compiler to reason about the source code and produced a vectorized version. ISPC includes other constructs that allow to compute efficiently with the SPMD model. An ISPC example is depicted in Figure .1 which shows what appears to be a C language program with extra reserved words. In this example, input variables are marked as *uniform* to indicate that these will hold the same value for all the vector lanes. The *foreach* construct indicates a parallel loop that will compute a different result for the variable *expected_y* and populate the *chi2* array with different values for each vector lane.

A key concept to ISPC are *gangs*. These are analogous to *warps* in the CUDA language, and are a group of program instances that run in parallel through the vector lanes. As shown in Figure 3 the way a *gang* interprets variables differs from it being *uniform* or *varying*. For

[1] A *warp* is a group of 32 threads that run the same instruction.

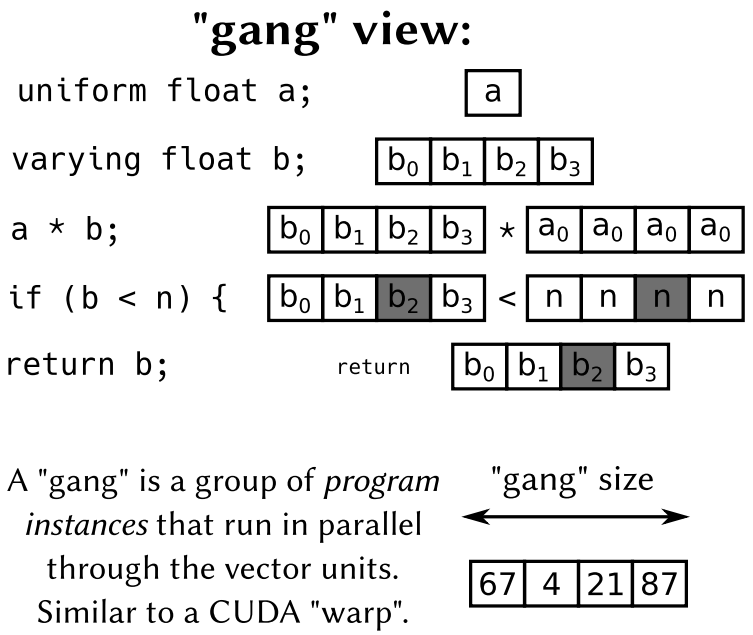

Figure 3. ISPC gangs

varying variables a *gang* will apply operations with the vector instructions for all the elements in parallel. This applies to control flow elements like an *if*, where if an element of the *gang* does not meet the condition it is masked to not take the result into account, even if it will be computed. ISPC adds a series of special constructs for control flow like *foreach*, *foreach_tiled* or *cif* among others. It is interoperable with C/C++ which allows to compile an algorithm with ISPC and use it with a C/C++ regular source code. It features compilation for SSE2, SSE4, AVX, AVX1, AVX2, AVX-512 and NEON instruction sets.

4 SPMD Compass tracking

The *Compass* algorithm is optimized for GPUs [4]. For this algorithm to run in a different architecture all GPU-specific optimizations need to be removed. *Compass* utilizes *shared memory* to cache hits that indicate the search windows used to find compatible hits. As CPU cache memory is not manually managed this optimization is removed and the main memory is used instead; hits are directly accessed as they are stored and the CPU will cache them. The ISPC compiler supports C language features with some extra extensions for the vectorization support, but C++ features are not supported. All C++ features used in the original algorithm are changed to be supported for the ISPC compiler, such as template metaprogramming used for various methods and types are specifically implemented for all the necessary cases that the templates were generating. Memory barriers used in the GPU to synchronize running threads and guarantee correct execution in parallel are not needed for the Compass algorithm; *gangs* run in parallel but these use the vector instructions of the processor which are forced to run all the elements at the same time, removing the need for synchronization in these cases. Memory barriers are offered by ISPC, but these are used to avoid data races between threads.

Gangs run in parallel using the vector lanes. Basic math operations are computed for all the elements, but when control flow structures are introduced inside a *gang* the flow of

the program can diverge leading to different instructions needed to be applied; the same problem applies for GPUs. ISPC compiler computes the values that do not meet a condition if at least one element of the *gang* meets it. For most cases this is not a problem other that the performance implications of divergence. For some cases as the instruction is actually computing the value for a case that did not meet a condition, these can lead to arithmetic exceptions. This cases are implemented specifically for the corner cases that showed during the *Compass SPMD* implementation were divisions by zero.

As a data-oriented algorithm, types that do not need to use more space than necessary in variables are in some cases saved as `half` types. The *UT Pre Decode* kernel benefits from this optimization, where the ISPC compiler offers support for `half` types with specific functions to convert from `float_to_half()` or `half_to_float` using the IEEE 16-bit floating-point format. The `half` type does not exist in the ISPC compiler and math operations over it is not possible, needing a conversion to `float` to operate and then converting back to `half`. A composed value is used to store two `half` types in a 16-bit type. For the ISPC compiler the supported types for this operation is `int16`, where the float 32-bit types can be stored in the regular way. Less precise functions are offered by ISPC as `_fast` functions, but for this implementation the higher precision result is preferred. A type casting issue is highlighted when doing this conversion as explicit type casts are needed when operating over the `int16` type holding the `half` value. For bit comparisons these need to be cast to `int32` to avoid precision problems.

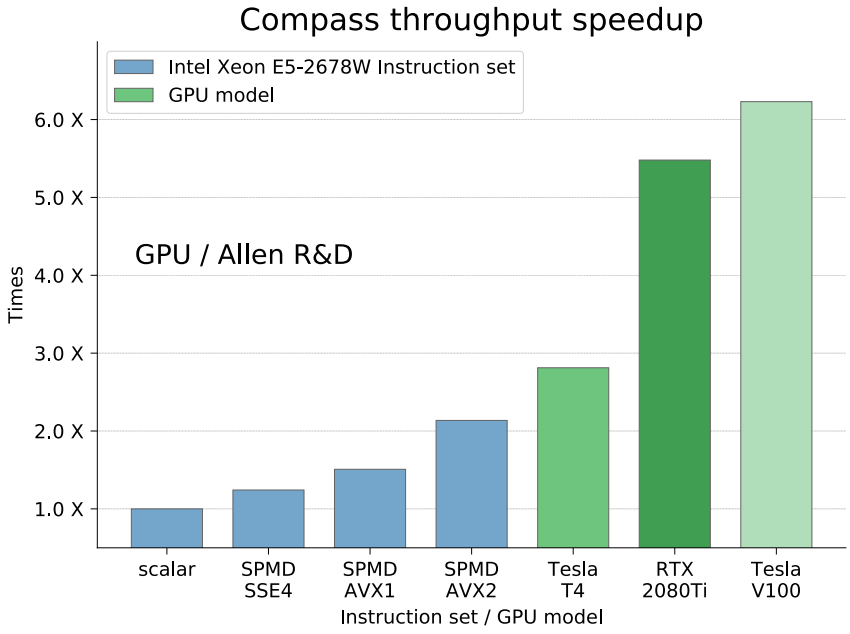

Figure 4. Performance comparison

All UT kernels are implemented with the SPMD model. Some kernels were found to run faster in the non-vectorized implementation compared to the SPMD vectorized implementation. Kernels with a naturally sequential algorithm, like the prefix sum, result in a faster computation being compiled with the GCC compiler. A selection of the faster kernels is used

to run the performance benchmarks that are shown in Figure 4. All instances run the same number of events and threads. An Intel Xeon(R) E5-2687X is used for the CPU measurements, where the GCC 9 and ISPC 1.12 are used as compilers for the different kernels. The figure shows the CPU measurements on the left in blue color; the base measurement shows the scalar implementation where all the kernels are compiled without ISPC. Different instruction sets targets are used to compared the performance benefits of these. More than 2× speedup is achieved with AVX2 target instruction set. Some performance is lost due to early track filtering and multiple branching, where the vector instructions are affected performance-wise. The usage of the *gangs* is lowered due to the candidate finding kernel where many conditional clauses are introduced and vector lanes are not used. The throughput achieved with different GPUs is highlighted for comparison: the GPUs achieve higher throughput for the Compass implementation.[2]

5 Conclusions and future work

UT tracking performance is improved by more than a factor 2 compared to the non-vectorized version. The Compass algorithm is used as a base, implementing it with the SPMD model to exploit the data-parallel characteristics of the original algorithm in CPU architectures. The resulting algorithm achieves better throughput while maintaining a highly readable and maintainability by using the ISPC compiler. It allows to write a vectorized algorithm without writing intrinsics or intermediate libraries, keeping a sequential-like source code that translates to vectorization. It allows to better reason about the algorithm and how it is parallelized for the vector units. Various levels of performance are achieved by compiling for different instructions sets, where the results match the expected increase in throughput with the increased vector width of the instructions.

Some SPMD kernels are not included in the throughput measurements and the sequential ones are used instead; these achieved better performance on the original version. As a future work extra speedup and vector lane usage may be achieved by fine tuning the algorithms to better meet the CPU architecture design.

6 Acknowledgement

This work has been partially supported by project "CABAHLA-CM: ConvergenciA Big dAta-Hpc: de Los sensores a las Aplicaciones" S2018/TCS-4423 from Madrid Regional Government.

References

[1] Collaboration LHCb (LHCb collaboration), Tech. Rep. CERN-LHCC-2012-007. LHCb-TDR-12, CERN (2012), `http://cds.cern.ch/record/1443882`

[2] R. Aaij, J. Albrecht, M. Belous, P. Billoir, T. Boettcher, A.B. Rodríguez, D.v. Bruch, D. Pérez, A.C. Vidal, D. Craik et al., Computing and Software for Big Science **4** (2020)

[3] M. Pharr, W.R. Mark, *ispc: A SPMD compiler for high-performance CPU programming*, in *2012 Innovative Parallel Computing (InPar)* (IEEE, 2012), pp. 1–13

[4] P. Fernandez Declara, D.H. Perez Campora, J. Garcia-Blas, D. Vom Bruch, J.D. Garcia, N. Neufeld, IEEE Access **7**, 91612 (2019)

[2] Allen SPMD source code is available at: https://gitlab.cern.ch/plfernan/Allen_SPMD