Compiler Support for an AI-oriented SIMD Extension of a Space Processor

Marc Solé, Leonidas Kosmidis

Barcelona Supercomputing Center (BSC) and Universitat Politècnica de Catalunya (UPC); email: {marc.solebonet, leonidas.kosmidis}@bsc.es

Abstract

In this on going research paper, we present our work on the compiler support for an AI-oriented SIMD Extension, called SPARROW. The SPARROW hardware design has been developed during a recently defended, award-winning Master Thesis and is targeting Cobham Gaisler's space processors Leon3 and NOEL-V. We present the compiler support we have included in two compiler toolchains, gcc and llvm as well as a SIMD intrinsics library for easy programmability. Compiler modifications are kept to minimum in order to enable incremental qualification of the toolchains. We present our experience working with the two compilers and performance results for the two compilers on top an FPGA implementation of the target space processor.

Keywords: compiler, SIMD, AI, space processor.

1 Introduction

In recent years, artificial intelligence (AI) and related topics, such as machine learning (ML) and neural networks (NN), have been explored in many different fields. Space systems are not an exception; the advantages that AI applications can provide in space operations are numerous, thus there are many on-going efforts to accelerate AI processing in space.

The simple, in-order, low-power processors traditionally used in space systems cannot meet the increased performance demands of AI. In such a critical environment, real-time capabilities and space qualification are mandatory properties, which are costly to provide in completely new designs.

While commercial off-the-shelf (COTS) AI accelerators and embedded GPUs have been used as an alternative in certain cases such as experimental missions and nano-satellites, they are not a definitive solution for high-risk missions. COTS accelerators are not radiation tolerant – a requirement to work beyond low-earth orbit – nor they have appropriate software stacks for the applications in space or support for real-time operating systems.

For this reason, in this recently presented Master's thesis project, we implemented SPARROW [1], a small, open-source SIMD module to accelerate the computation of AI applications in an already qualified, widely used space processor, LEON3, with minimal hardware and software changes. It is directly connected into the integer pipeline and provides additional vector instructions to improve such applications.

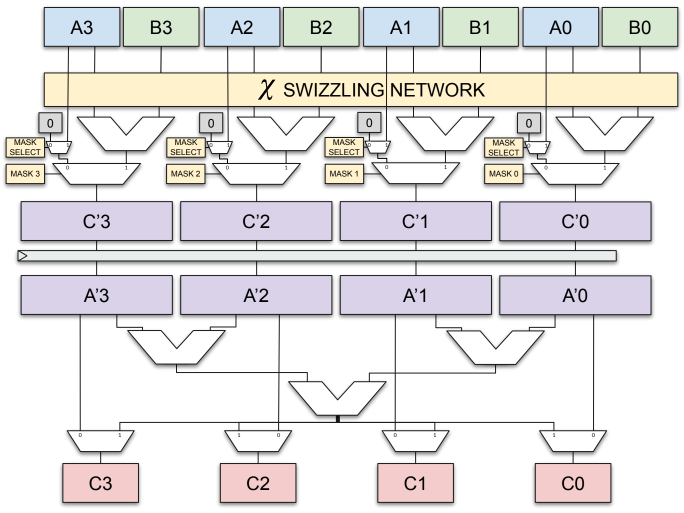

Figure 1: Overview of the SIMD SPARROW module [1]

The hardware cost of the module is minimal compared with conventional vector approaches, thanks to the re-utilization of the integer register file. This is possible since 8-bit operations have been shown enough for AI applications in the literature and in commercial AI hardware. Therefore, each integer register can work as a vector with up to 4 8-bit components. To our knowledge, this is a unique feature of our work. Further advantages of our choice is the simplification of data management, which eliminates the need for new load-store instructions, allowing a small incremental qualification cost of the hardware and its compiler. More details on the hardware design of the module are provided in [1].

In this work in progress paper, we describe our on-going work regarding the addition of software support for SPARROW with two widely used compilers, gcc and llvm. We describe our experience working with these two compilers and the development of small preprocessor library which allows to program SPARROW in a similar way with SIMD intrinsics for other processors. In addition, we provide some early comparison results of the performance of the two compilers both with handwritten assembly implementations as well as with our SIMD library.

2 Background on SPARROW Design

Before we discuss about the compiler support, we first need to briefly describe the SPARROW hardware design. The SPARROW SIMD unit is co-designed analyzing the most important features and characteristics of ML workloads. The

```
 1 unsigned char weights[32*32];
 2 unsigned char next_layer [32*32];
 3 unsigned int a, b, result , scr;
 4
 5 /* set the value of the %scr */
 6 scr = 0x0D9D0E; //swizzling_B = 1-2-3-0,
 7                 // swizzling_A = 3-2-2-0,
 8                 // mask_select = 0, mask = 1110
 9
10 asm("wr %0", %%scr : : "r"(scr));
11 /*  initialise  all a components to 0, ie a.xyzw=0 */
12 a = 0;
13 /* b.xyzw = weights [0]. xyzw */
14 b = *((unsigned int*) &weights[0]);
15 asm("nop"); // wait for %scr to commit the write
16 /* result .xyz = a.xyy + b.zyx */
17 asm("usadd_ %1, %2, %0": "=r"( result ) : "r"(a), "r"(b));
18 /* next_layer [0]. xyzw = result .xyzw */
19 *((unsigned int*) &next_layer[0])=result ;
```

Figure 2: Example of SPARROW programming in C with inline assembly

dot product is one of the most frequently used computations in ML as it is used in matrix multiplication which is a recurrent kernel in the computation of NN for fully connected layers and convolutions. SPARROW features a 2-stage approach which allows to compute a dot product with a single instruction as can be seen in Figure 1. In the first stage, for vector-vector operations, the data are computed in parallel allowing up to 4 simultaneous operations. In the second stage, reduction operations are performed on the result from the first stage. Both stages can be bypassed in order to just perform any of the two types of operations. This allows up to 200 combinations of operations such as addition, multiplication, maximum and minimum, bitwise operations, etc with just the introduction of 13 vector instructions in the first stage and 4 reduction operations in the second one. Both signed and unsigned version are included.

Additionally, SPARROW includes GPU-like features to provide even more flexibility to the module, such as masking and swizzling. Both characteristics can be controlled by using a special register, the SPARROW control register (%scr). As other special registers it can be accessed using the already existing instructions in the ISA. To avoid overflow, SPARROW also includes a saturation version of the instructions, both signed and unsigned, which clips the results at 0 to 255 for unsigned or at -128 to 127 for signed.

The module is highly portable and can be used in different base processors with minimal modifications. Currently SPARROW has been integrated with the SPARCv8-compliant LEON3 processor, developed by Cobham Gaisler, preserving its 100MHz frequency. The module has also been ported to the NOEL-V processor, a RISC-V based space processor by Cobham Gaisler, as we discussed in our talk at the RISC-V Forum: Vector and Machine Learning [2] [3]. SPARROW won the fist position in Xilinx's Open Hardware Competition 2021 in the student category, and awarded the best Master Thesis in Spain for 2021 by the Spanish IEEE AESS Chapter.

3 SPARROW Software Support

3.1 Compiler support

An important advantage of SPARROW compared to custom accelerators is the ability to reuse the existing qualified software stack of LEON3 i.e. the RTEMS real-time operating system or bare-metal space applications, which reduces both the cost and the effort of the development of a new compiler from scratch as well as its qualification cost later.

```
 1 unsigned char weights[32*32];
 2 unsigned char next_layer [32*32];
 3 unsigned int a, b, result , scr;
 4
 5 /* set the value of the %scr */
 6 __sparrow_setMask(0b1110);
 7 __sparrow_setMaskSel(0);
 8 __sparrow_setSwizzlingA (3,2,2,0) ;
 9 __sparrow_setSwizzlingB (1,2,3,0) ;
10
11 __sparrow_writeSCR();
12 /*  initialise  all a components to 0, ie a.xyzw=0 */
13 a = 0;
14 /* b.xyzw = weights [0]. xyzw */
15 b = *((unsigned int*) &weights[0]);
16 asm("nop"); // wait for %scr to commit the write
17 /* result .xyz = a.xyy + b.zyx */
18 __nop( result , a, "usadd", b);
19 /* next_layer [0]. xyzw = result .xyzw */
20 *((unsigned int*) &next_layer[0])=result ;
21
```

Figure 3: Example of SPARROW programming in C for SPARC v8 with the SPARROW SIMD library

We added SPARROW support in the two most widely used compilers nowadays, gcc and llvm. We modified the `binutils` of Gaisler's `bcc-2.2.0` gcc-derivative compiler and the base `LLVM v13.0`. We use an underscore as separation between the instruction names of the two stages. In the case of `nop` it is omitted for the second stage. An `s` and `u` prefix in the instruction name denotes saturation and unsigned operation, i.e. `usmul_`. We also added aliases such as the dot product, which can be both represented by `mul_sum` or `dot`. The SPARROW control register can be accessed using the `wr`, `rd` and `mov` instructions present on the SPARC v8 ISA for accessing special registers.

We program SPARROW in C, using inline assembly instructions as shown in the example of Figure 2 for a saturated vector addition with unsigned 8-bit values using swizzling and masking. As it can be seen, SPARROW can be programmed in a high level way, not very different than vector intrinsics for conventional SIMD extensions such as NEON.

Another important advantage of reusing the integer register file is that we can use the regular load and store instructions. Inline assembly (and code generation) is only required for the SIMD operations. Moreover, this means that we don't need to specify explicit registers in the inline assembly, nor to modify the compiler register allocator. Notice that by passing the integer variable names to the inline assembly instruction (line 17), the SIMD instruction accesses directly the register in which each of the variable is allocated by the compiler.

3.2 SPARROW SIMD Library

A disadvantage of programming SPARROW in assembly is that there are features like the mask and swizzling which the programmer needs to be aware of. In order to make the setting of the SPARROW Control Register transparent, we decided to create a library that contains multiple definitions to simplify working with SPARROW in SIMD intrinsics fashion.

Function	Description
`__sparrow_readSCR(X)`	Stores the current value of the SPARROW Control Register in the variable `X`
`__sparrow_writeSCR()`	Writes in the SPARROW Control Register the value of `__sparrow_scr`
`__sparrow_set(X,Y)`	Writes in the SPARROW Control Register `X xor Y`
`__sparrow_resetSCR()`	Resets the value of the SPARROW Control Register
`__sparrow_setMask(X)`	Sets the mask bits of `__sparrow_scr` to `X`
`__sparrow_setMaskSel(X)`	Sets the mask selection bit of `__sparrow_scr` to `X`
`__sparrow_setSwizzlingA(X,Y,Z,W)`	Sets the first operand swizzling order in `__sparrow_scr` to `X-Y-Z-W`
`__sparrow_setSwizzlingB(X,Y,Z,W)`	Sets the second operand swizzling order in `__sparrow_scr` to `X-Y-Z-W`
`__sparrow_(op1, op2, A, B, C)`	Performs the `op2` reduction on `A op1 B` and stores the value in `C`
`__nop(C, A, op1, B)`	Computes `C = A op1 B`
`__sum(C, A, op1, B)`	Computes a sum over `A op1 B` and stores the result in `C`
`__max(C, A, op1, B)`	Computes the maximum in `A op1 B` and stores the result in `C`
`__min(C, A, op1, B)`	Computes the minimum in `A op1 B` and stores the result in `C`
`__xor(C, A, op1, B)`	Computes a xor reduction over `A op1 B` and stores the result in `C`
`__usum(C, A, op1, B)`	Computes an unsigned sum over `A op1 B` and stores the result in `C`
`__umax(C, A, op1, B)`	Computes the unsigned maximum in `A op1 B` and stores the result in `C`
`__umin(C, A, op1, B)`	Computes the unsigned minimum in `A op1 B` and stores the result in `C`

Table 1: SPARROW library functions

The SPARROW SIMD library is implemented using C-preprocessor macros that convert function-like calls into the inline assembly. For the SCR, a variable is declared which is modified when setting the mask and swizzling and is used to write in the special register. One of the advantages of having a library implemented like this is once again portability and simplicity. Table 1 shows the existing functions in the SPARROW library.

In Figure 3 the same code shown in Figure 2 is represented using the SPARROW library. Note that the setting of the SCR, which starts at line 6, requires more instructions, however since those are C-preprocessor macros the compiler can reduce the number of actual generated instructions. On the other hand, although the same behaviour as with the inline assembly could be achieved by using `__sparrow_setSCR(X,Y)`, with this approach the value of each field is more clear and the programmer does not require any knowledge on the SCR organization. In line 11 it is necessary to include the writing of the SCR as the previous lines were just setting the library internal variable. This is done to reduce the number of accesses which otherwise, would be necessary if each line performed the actual write.

4 Preliminary Experimental Results

We have evaluated our compilers and SIMD library on an FPGA implementation of SPARROW integrated with LEON3 [4]. For our preliminary evaluation we are using a widely used and well understood kernel, matrix multiplication, which is an essential block for AI inference applications, since it is used for representing fully connected layers as well as for the implementation of convolutions. In addition to the sequential version of matrix multiplication written in C which is used as a baseline, we have produced two additional versions of the code, one written in SPARROW assembly and one using the SPARROW SIMD library. Moreover, we have developed two variants of matrix multiplication, one using saturation and one allowing the values to wrap around when an overflow occurs. However, due to space reasons we only provide results with the version with saturation, since the results for the other version exhibit the same trends, but with lower speedup ranges (from 2.1× - 6.8×). All programs are compiled using the highest optimisation level (-O3).

Figure 4 shows the comparison between the various versions, for different size of matrices, for common matrix sizes found in machine learning applications. The results are normalised with respect to the gcc sequential (CPU) version, which is also shown in the figure with a red line at value 1. As a consequence, higher values are better and values over the red line represent speedup, while values below it show slowdown.

First we compare the sequential versions of the two compilers. For the two smaller sizes (4 and 8), gcc provides slightly faster code than llvm, while for sizes 16 and 32 llvm is faster. However, when the size of matrices exceed the size of the data cache (8KB), the performance difference is negligible.

When comparing the SIMD assembly versions, we notice that gcc generates faster code than llvm for all sizes, achieving a maximum speedup of 17.3× over the sequential version for matrix size 32. The llvm still provides a good speedup compared to the sequential version, up to 15.2×, being only 10% slower that gcc.

The SIMD library inevitably incurs some overhead compared to the assembly implementations, especially in the case of gcc. However, llvm provides the same performance with the version that uses assembly instead of the library for sizes of 128 and larger. Finally, comparing the SIMD library versions on top of gcc and llvm, gcc provides the same performance with llvm, except in sizes 8 and 16, where it is slightly faster.

5 Lessons Learnt

Having worked with both GCC and LLVM for the development of the software support for SPARROW has allowed us to compare, not only the performance, but also the experience when working with each one. It is worth noting that we had no prior experience on working on either of the two toolchains

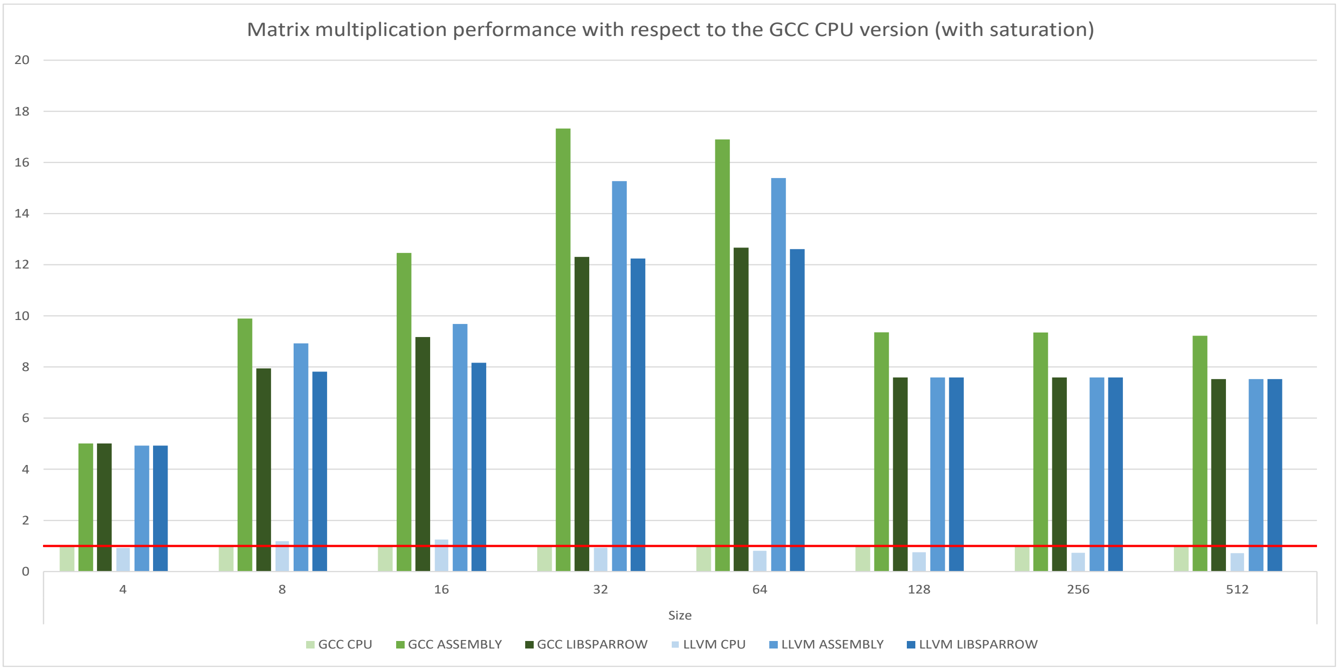

Figure 4: Plot of the performance for matrix multiplication with saturation enabled

prior to this project. In the previous section we already presented a preliminary performance comparison between the two compilers. A detailed analysis would require more experimentation and the evaluation in different scenarios. In general, however, it is shown that gcc-compiled executables have lower execution times, at least for LEON3.

When working to include the SPARROW assembly instructions one of the key advantages of llvm over gcc was the possibility of defining the instructions in a nested way. This simplifies the addition of two-stage instructions allowing a simpler combination of them. However, a new line for each combination must be manually added. On the other hand, the code for adding these instructions is easier to understand in GCC, which can be easily deduced from the existing instructions. Fortunately, LLVM has a great documentation and a large number of tutorials about on how to modify it.

All in all, both compilers had advantages and disadvantages compared to one another, however they both offer good facilities to implement the required functionality in order to add software support in new hardware designs.

6 Conclusions and Future Work

In this paper we have presented our on-going work on the addition of software support for the SPARROW SIMD unit in the gcc and llvm compilers, and a SIMD library that facilitates SPARROW programming without using inline assembly. In terms of development both compilers had advantages and disadvantages compared to one another, however they both offer good facilities to implement the required functionality in order to add software support in new hardware designs.

In terms of performance, we noticed that gcc provides higher performance when sequential C code or assembly is used, but both compilers provide similar performance when our SIMD library is used. As a future work we want to perform an extensive evaluation of the compiler backends we developed as well as of our SIMD library, by porting more applications in SPARROW. Moreover, we would like to evaluate Ada's frontends of both compilers, generate an Ada version of our SIMD library and compare them among them and with the C frontends. Finally, we plan to add support in the compilers so that they can generate directly SPARROW assembly instructions, through autovectorisation.

7 Acknowledgments

This work was funded by the Ministerio de Ciencia e Innovacion - Agencia Estatal de Investigacion (PID2019-107255GB-C21/AEI/10.13039/501100011033 and IJC-2020-045931-I) and partially supported by the European Space Agency (ESA) through the GPU4S (GPU for Space) activity and the HiPEAC Network of Excellence.

References

[1] M. Solé and L. Kosmidis, "SPARROW: A Low-Cost Hardware/Software Co-designed SIMD Microarchitecture for AI Operations in Space Processors," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2022.

[2] Linux Foundation, "RISC-V Forum: Vector and Machine Learning." https://events.linuxfoundation.org/riscv-forum-vector-and-machine-learning.

[3] Marc Solé, Leonidas Kosmidis, "RISC-V Forum: Vector and Machine Learning." https://events.linuxfoundation.org/riscv-forum-vector-and-machine-learning/program/schedule/.

[4] M. Solé and L. Kosmidis, "SPARROW source code repository," 2021. https://gitlab.bsc.es/msolebon/sparrow.