A C COMPILER FOR THE WIDE, LOW-POWER CONNEX-S VECTOR ACCELERATOR

Alexandru E. Şuşu[1]

Connex-S is a wide vector accelerator designed at the Electronics department at Politehnica University of Bucharest. It is a competitive customizable architecture for embedded applications with 32 to 4096 16-bit integer lanes.

Our C compiler targets OPINCAA, a JIT vector assembler and coordination C++ library for Connex-S accelerating computation for an arbitrary CPU. Therefore, we address aspects of automatic parallelization such as efficient vectorization, communication, and synchronization. We also develop the back end for Connex-S and support for emulation of arithmetic operations for unsupported types such as 16-bit floating-point and 32-bit integer.

We discuss various optimizations we perform to compile efficient OPINCAA code and describe why the generated code is correct.

We report speedup factors of at most 12.24 when running on a Connex-S accelerator with 128 lanes w.r.t. the dual-core ARM Cortex A9 host clocked at a frequency 6.67 times higher. Also, we achieve a modest energy efficiency improvement average of 1.1 times. However, note that a Connex-S dedicated integrated circuit can achieve an order of magnitude more energy efficiency than our FPGA implementation.

Keywords: Connex-S vector accelerator, low-power wide SIMD, vectorization, vector-length agnostic compiler, LLVM, symbolic compiler memory allocator, quantitative static analysis, source-to-source compiler, emulation of arithmetic operations, energy profiling

MSC2010: 53C 05.

1. Introduction

Data parallelism is an optimization strategy that divides the data domain of a problem into multiple regions and assigns different processors to compute the results for each region [18]. Data-parallel programming is a sweet-spot [14]: parallel architectures can support it well and a lot of algorithms can be described in this manner. Data parallelism is best matched by *SIMD (Single Instruction Multiple Data)* computer architectures. SIMD processors achieve among others better energy efficiency because they use considerably fewer resources for the control unit [41] and the compiler is delegated to extract ILP instead of the hardware of a superscalar processor. Vectorization support for SIMD architectures is better these days, available in open-source compilers like GCC and LLVM, besides proprietary alternatives like Intel ICC and IBM XL. This is why mainstream processor manufacturers add more SIMD units per chip and increase their widths: ARM recently introduced the *Scalable Vector Extension (SVE)*, which is an instruction set operating on maximum 2048 bits wide data for the ARMv8-A architecture, Intel's Xeon Phi manycore processor integrates up to 72 x86 cores with *AVX-512 (Advanced Vector eXtensions)* handling 512-bit vectors, NVIDIA's *Graphics Processing Units (GPUs)* incorporate an increasing number of SIMD units at each newer generation and so on.

In this paper, we present an LLVM [30] compiler for a system using the Connex-S vector accelerator. Connex-S is a customizable processor with 32 to 4096 16-bit integer lanes, currently implemented in *FPGA (Field-Programmable Gate Array)*, loosely integrated with a CPU, which in this paper is ARM Cortex A9, suitable for embedded systems applications acceleration, being low power [12, 45]. Connex-S is a *wide* vector processor, by which we

[1]PhD student, ETTI, UPB, Romania, e-mail: alex.susu@gmail.com

understand it has more than 32 lanes. Our compiler is *vector-length agnostic (VLA)*, which means that the generated code can run on Connex-S accelerators of different widths (also called vector lengths).

We choose LLVM [30] because it is an open-source compiler, with a better design than GCC, with a well-defined *Static Single Assignment (SSA)* based intermediate representation (IR) and a thriving community.

We perform offloading to the accelerator by using OPINCAA, a *JIT (Just-In-Time)* vector assembler and coordination C++ library for Connex-S [13], which allows running portably C++ host code together with easily readable, vector-length agnostic Connex-S assembler programs. Note that, in principle, OPINCAA can also make the vector code more efficient by easily specializing it with techniques such as loop unrolling during the JIT assembling.

Our Connex-S processor is designed to be low-power compared to existing architectures such as NVIDIA GPUs. Current NVIDIA GPUs suffer from power inefficiency issues due to: i) small SIMD units with a maximum of 32 lanes (EUs) because their legacy is to accelerate 3D computation and the bus accessing the RAM has a width of up to 384 bits; ii) big register files required to allow switching easily context in order to overcome the limited memory bandwidth [36]; iii) the GPU runtime schedulers, which offer easiness of programming with the CUDA SPMD programming model, which is expressive and easy to compile at the expense of a runtime overhead to schedule the code; iv) complex controllers with conditional branch reconvergence. All these points are addressed by our wide Connex-S accelerator, especially by its high bandwidth local banked vector memory, and its auto-vectorizing LLVM compiler that performs many tasks that otherwise would be performed at runtime. This has the potential to save energy. We perform in this paper a few experiments to analyze the energy efficiency of the Connex-S accelerator w.r.t. its host. Note however that our processor is currently implemented in a Xilinx FPGA, and FPGAs are an order of magnitude less power efficient than a dedicated *integrated circuit (IC)* fabricated in the same silicon technology [28, 33].

The contributions of this paper are:

- the compilation from a sequential C program to *VLA (vector-length agnostic)* code. To achieve this goal the back end we write for the Connex-S vector processor has special support for *symbolic* scalar immediate operands, which are to be handled by the OPINCAA JIT vector assembler. Note that this allows passing source program variables from the CPU scalar memory to the vector kernel.
  Also, we need to adapt LLVM's loop vectorizer pass in order to perform symbolic quantitative static analysis to retrieve the trip counts of loops and the number of source code array elements accessed in our vectorized code, and possibly recover them as symbolic expressions from LLVM IR to C/C++. This quantitative analysis allows us to provide input to our OPINCAA function calls requiring the amounts of data transferred at runtime, its JIT assembler using e.g. symbolic trip counts for vector strip-mining, and our *symbolic compiler memory allocator*.
- experimental results using our compiler on a test suite of simple C programs, namely the native 16-bit integer type and emulated 32-bit integer and 16-bit floating-point types. Besides measuring the execution time for the benchmarks when running on the CPU or on the Connex-S accelerator, we perform also energy measurements in the two scenarios for all programs.

Our approach is to extend existing open-source tools, such as LLVM, to achieve correct and efficient source-to-source compilation from sequential C to coordinated OPINCAA programs with portable CPU C++ and Connex-S vector assembly code.

The paper is structured as follows. In Section 2, we present some of the most relevant works similar to ours. In Section 3, we detail the architectural features of the Connex-S vector accelerator. Section 4 describes the *Instruction Set Architecture (ISA)* of the Connex-S processor, its OPINCAA programming model, and the semantic gap between the Connex-S assembler and the LLVM IR languages. Section 5 discusses the most relevant features of the Connex-S OPINCAA LLVM compiler. In Section 6, we present experimental results regarding execution time and energy consumption.

2. Related Work

Autovectorization, the compilation for a SIMD target, is an established 40-year-old research topic [43, 5]. Nuzman and Zaks are the first to implement it in the GCC compiler [38, 39]. Eichenberger et al. implement support for the vector units of the Cell processor inside the IBM XL compiler [19]. Vectorization is normally included as a middle-tier pass, but exceptions exist, such as the Scout source-to-source transformation tool [27], which generates directly inside the C code intrinsics for SIMD instruction sets such as Intel AVX, SSE or ARM NEON. The mainstream adoption of GPU accelerators has motivated the creation of many programming models for them. Recently, a pragmatic approach is to compile directly sequential C to such parallel architectures. The polyhedral compilation of C to explicit parallel programming languages such as OpenCL or CUDA is presented by Grosser et al. in [22]. Our work is similar to theirs, but we target only Connex-S and, therefore, have to focus on the challenging issues of the customizable architecture.

Recently, the LLVM compiler [7, 20] for the ARM *Scalable Vector Extension (SVE)* instruction set generates vector-length agnostic binary programs [46], but tight integration with the ARMv8-A CPU requires no coordination of the SVE unit since it shares, among others, the RAM with the CPU. For vector-length agnosticism, they require: i) to add special instructions to the ARM SVE ISA, such as **inc*** (increment a scalar register by the number of elements in an SVE vector), **index**, **while*** [46]; ii) to add a scalable attribute for LLVM IR vector types and add a few new IR instructions, such as *vscale*, *stepvector*, *propff*, and extend *shufflevector* to accept non-constant masks [7]); iii) they introduce separate stack regions for scalar and vector values, the latter being dynamically allocated, in order to support vector spills, fills and argument passing [46]. Our Connex-S architecture supports easier vector-length agnosticism because it has a local banked vector memory of width CVL, hence we do not need: instructions like **inc***, nor stack regions. On the other hand, we also have an instruction similar to **index**, which we call **ldix**.

Similarly, the RISC-V ISA [49, 24] has vector-length agnostic extensions [48, 10]. The vector ISA can change dynamically the vector length, the number and element type of vector registers. For example, it allows doing strip-mining with vector residual loop by using the RISC-V vector instruction *vsetvl*.

Nuzman et al. address the generation of portable SIMD code by using GCC to generate vector .Net code for the Mono VM, which is then able to run the code efficiently on an x86 with SSE or a PowerPC with AltiVec SIMD unit [40]. Our generated OPINCAA programs with C++ host code and Connex-S vector-length agnostic assembler code are portable mainly because of our low overhead JIT assembling technique, which can access the CVL variable and execute a compiled vector loop in order to match the trip count of the original strip-mined loop.

A difficult subtask is to automatically generate communication primitives between the CPU and the accelerator. For this, the DawnCC compiler [35], performs static *Symbolic Range Analysis (SRA)* in order to offload data of parametric size to an accelerator via the OpenACC and OpenMP 4.0 offload language extensions. We make use of this static analysis in our project.

The advantages of banked vector memories, such as the local memory of Connex-S, are discussed in many papers like [29, 11, 32] and the book by Akl [4]. Armejach et al. [8] draw attention that when running 3D stencil applications the standard CPU memory hierarchy can become a performance bottleneck for the ARM SVE SIMD unit when it is wider than 512 bits. Similarly, Hennequin et al. discover the same thing for Intel's x86 AVX-512 SIMD unit [23].

3. The Connex-S Vector Accelerator

The Connex-S processor is meant to be loosely integrated with a CPU, following the accelerator paradigm.

Connex-S is essentially a SIMD processor with a number of 16-bit integer *execution units* (EUs, also called *lanes*, *cells*, or *Processing Elements*; an EU can also support operations of different types than 16-bit int, in principle) fixed at design time between 32 and 4096, for which we assign variable *CVL, ConnexVectorLength*, normally a power of two, and a hardware reduction tree. The processor is designed to support well kernels found in *Basic Linear Algebra Subprograms (BLAS)* libraries.

We depict the architectural organization of a system with Connex-S accelerator and an arbitrary CPU in Figure 1.

Connex-S is a Harvard architecture, with separate *Internal Instruction Memory (IIM)* [1] and *Local Storage (LS)* memory. The processor has a predictable performance because all instructions have basically the same latency and it has no sources of unpredictability such as cache misses or a branch misprediction. We offload on the Connex-S accelerator *intensive* computations [34], which have a simple, predictable execution; this is why Connex-S does not need a data cache memory, which can address in part more difficult to predict memory accesses.

Connex-S has an in-order pipelined controller with the following four stages: i) fetch instruction from the IIM; ii) read registers and decode the instruction; iii) execute the instruction or calculate an address; iv) write the result into a register. Note that the decoded control signals also need to go through the distribution network of the controller, which has a latency of $log_2(CVL)$ cycles.

Using 16-bit data paths is justified by the facts it reduces power consumption and many integer benchmarks require only words of 16 bits or fewer [15, 9]. The processor does not include hardware support for floating point because it is costly. Adding support for 32-bit floating-point add, subtract, compare, and multiply operations is reported by Karuri et al. [26] to increase the area of a chip by 2.5 times. On the other hand, the paper claims floating-point emulation is one order of magnitude less energy efficient than direct hardware execution. Therefore, since our wide processor can reach up to 4096 lanes, we choose to emulate floating point in order to have a considerably smaller processor area. We present in [16] how we perform efficiently floating-point emulation by delegating the LLVM compiler to inline the emulation code and perform standard optimizations on it such as register allocation.

In its largest instance, with 4096 lanes, Connex-S is basically two orders of magnitude larger than the widest SIMD units handled by: i) ARMv8-A's *SVE* instruction set, with 2048-bit vector operands, ii) Intel Xeon Phi's *AVX-512*, with 512 bits long vector units for each core and iii) NVIDIA GPUs, with SIMD units of at most 32 lanes of 32 or 64 bits each.

For this paper, we implement Connex-S inside a Xilinx Zynq-7020 FPGA [42]. The number of EUs of Connex-S can be easily changed in the Verilog description of the processor to match the needs of the running application. In principle, we can also change in the Verilog code the size of the LS memory or of the IIM, the number of vector registers or add special

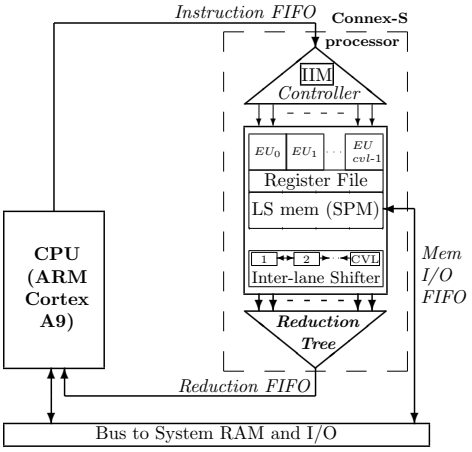

FIG. 1. The architectural organization of a system with a Connex-S vector accelerator

functional units to execute efficiently operations like the scan (prefix) higher-order function. This customization is part of a process called hardware/software codesign [17].

The 28 nm Zynq-7020 SoC also has a dual-core ARM Cortex A9, so this platform can be used for embedded applications, such as computer vision algorithms. This Connex-S implementation with 128 lanes consumes a maximum of 1.5 Watts at 100 MHz, while Cortex A9 consumes up to 0.8 Watts at 668 MHz in our experiments. But a 28 nm IC version of Connex-S with 128 lanes at 1 GHz consumes lower than 0.5 Watts [33], which is more than an order of magnitude more power-efficient.

The LS memory of Connex-S is a *scratchpad memory (SPM)*, separate from the CPU RAM. Therefore, the programmer or the compiler need to issue memory transfers, which employ a *Direct Memory Access (DMA)* unit, to offer Connex-S the operands it needs to work on and, optionally, to copy back data from the LS to the CPU RAM. The Connex-S LS memory is a banked vector memory, being a memory that has an invariable, predictable one cycle access latency. The Connex-S LS memory allows each lane to access only its content. Communication between the lanes is performed efficiently and predictably using an inter-lane shift unit. Following the classification from Akl [4], this makes Connex-S an *interconnection network* (as opposed to *shared memory*) SIMD computer with a *linear array* topology. Connex-S has only one *inter-lane shift* register in which we can load any vector value, in which the lanes can communicate in one step with their direct neighbors, similarly with the cells of systolic arrays [6].

When synthesized in a Xilinx Zynq-7020 FPGA, the LS memory has normally 1024 lines with 128 16-bit elements, given the limited number of gates of the FPGA. To these, we also add normally another 200 lines exclusively dedicated for vector register spilling and for storing tables for floating-point arithmetic operation emulation. Also, the IIM has normally a capacity of 4 KB.

Connex-S has a RISC-like ISA [21] with a hardware loop mechanism with a scalar counter and block predication but has neither call nor general branch instructions. Connex-S is little-endian.

Connex-S has no scalar data memory. The 32 Connex-S vector registers, with a number of CVL, normally 128, 16-bit integer elements, are named in OPINCAA $R(0)$, ..., $R(31)$, and the line i of the LS memory is referred to as $LS[i]$.

Connex-S communicates with the rest of the system through its instruction, memory I/O (inbound and outbound memory transfers between CPU RAM and LS memory), and reduction FIFOs, as depicted in Figure 1, all of them having a capacity of 512 32-bit words.

We can make these FIFOs perform faster transfers by increasing the word size to 128 bits, for example, which also leads to bigger power consumption. Normally, Connex-S receives its operands in the LS memory via I/O transfers from the system RAM and returns results only via the reduction FIFO, without performing I/O reads from the LS memory.

The I/O calls of Connex-S have a big overhead: for example, sending 256 or 512 bytes from Connex-S takes basically the same amount of time of 72 μs. Therefore, to increase the throughput of the I/O transfers, we need to aggregate, where possible, the copies for each array, in order to mitigate the big overhead of a transfer. However, as already discussed, there is one difficulty in doing this, namely, we need to know the length of the array being transferred. We compute the length statically by using the *SRA* LLVM pass.

Connex-S can be used as an educational processor like the Vector DLX [11], especially because it has a clean ISA and an easy to use assembler tool.

The synthesized Connex-S processor at 100 MHz, with 128 lanes achieves a large LS memory bandwidth of 26 GB/sec, comparable with the speed of the latest DDR5 desktop PC RAM modules available on the market, putting in evidence the performance advantage of the LS banked vector memory normally with 1224 words (or 2.4 KB) per lane.

4. The Connex-S ISA and Its OPINCAA Programming Model

The Connex-S *Instruction Set Architecture (ISA)* contains pure SIMD operations like arithmetic, bitwise logical, logical, memory access, and **nop** instructions. It also has special vector instructions: load immediate operand (**vload**), sum-reduce (**red**, which takes $log_2(CVL)$ cycles to execute), inter-lane shift operations (**cellshl/r** and **ldsh**), which basically move data between lanes one position per cycle, block predication instructions (**whereeq/lt/cry** and **endwhere**, which in OPINCAA are called *EXECUTE_WHERE_EQ/LT/CRY/_IN_ALL,*) and simple loop with counter instructions (**setlc** and **ijmpnzdec**, in OPINCAA represented by *REPEAT(imm)* and *END_REPEAT*, which currently do not allow loop nesting and have a body size limited by the capacity of the *IIM*).

The **where** blocks are useful because: i) they can reduce the instruction bit length since the predicate register is not encoded in it; ii) they send fewer decoded control signal bits to each lane for each predicated instruction, which has the potential for saving energy.

A complete description of the ISA is available in [21]. Also, we can find in Table 2 from Section 6 all the instructions of the vector accelerator together with their energy consumption. As we can see, Connex-S has a few control flow instructions allowing to run normally only non-nested loops of constant trip counts and to use a predication mechanism using the Boolean values of the Carry, Less or Equal flags, set previously for each lane. It does not have call or conditional branch instructions, available, for example, in NVIDIA GPU's PTX assembly [41].

All the instructions take vector operands of 16-bit signed integer (*i16*) elements, unless otherwise specified, and a few of them can have an immediate operand. We also require having 16-bit unsigned integer (*u16*) instructions for multiplication and reduction, **mult.u16** and **red.u16**, which help for the efficient emulation of reduction and multiplication operations for 32-bit (or larger) signed integers. Note that the **mult.i16/u16**, accompanied by the result reading instructions **multlo** and **multhi**, use the DSP48E1 functional units of Xilinx Zynq, which can perform efficiently, among others, i16 or u16 multiplication.

4.1. The OPINCAA Programming Model

OPINCAA, an acronym for OPcode INjector for the ConnexArray Architecture [13], is an everything (decomposition, mapping, communication, and synchronization) explicit parallel programming model [44] for a system with one Connex-S accelerator.

The OPINCAA JIT vector assembler and coordination C++ library for Connex-S does the following: i) it allows writing, sequential *C++ host code*, together with easily readable and modifiable Connex-S vector-length agnostic assembly kernels directly in C++ by overloading the standard C++ arithmetic operators - this makes the syntax of the Connex-S assembly simple to learn; ii) it assembles at runtime on the host the kernels to be dispatched for execution on Connex-S; iii) it caches the JIT assembled instruction binary stream of a kernel to increase the performance of its future invocations; iv) it performs coordination for the accelerator, such as communication and synchronization between an *arbitrary* CPU and Connex-S.

OPINCAA can assemble at runtime instructions with *symbolic* scalar immediate operands from C/C++ expressions, representing, for example, the sizes of arrays, loop trip counts, or the width of the vector processor. In effect these make the OPINCAA programs vector-length agnostic, meaning that we can run them on Connex-S machines of arbitrary width, mostly due to the program environment variable CVL and the CVL-parametric strip-mining. Also, since Connex-S has a limited hardware loop mechanism, we can write in OPINCAA *host-side for* loops, which are simple C++ for loops that unroll the assembly code inside their bodies, which helps to implement loop nests of arbitrary depth.

Given the preceding arguments, we choose that the result of our compiler is an OPINCAA C++ program.

OPINCAA is similar to OpenCL [37] in the sense both coordinate execution in heterogeneous systems and employ some form of runtime code generation, but OPINCAA targets just the Connex-S accelerator and writes kernels for it in vector assembly language instead of OpenCL C. OPINCAA is more efficient normally than OpenCL because it does not compile at runtime the kernels for the accelerator, making it more appropriate for embedded systems applications. Note that the specialized *embedded profile* of OpenCL fixes this inefficiency.

5. The Connex-S OPINCAA LLVM Compiler

Now that we introduced the heterogeneous platform we target with the Connex-S accelerator, we start describing our LLVM-based compiler taking sequential C source programs.

Following the terminology of parallelization steps introduced in Skillicorn et al. [44], the compiler has to decompose the program into tasks, map them on the EUs, synchronize them and establish communication between tasks. Fortunately, for SIMD architectures synchronization is not an issue due to the lock-step fashion they operate [11].

We present in Figure 2 a flowchart with the stages of our compiler. We first parse the C source file with the *clang* command, which generates unoptimized LLVM IR code. Then, we run the *opt* command, which optimizes the LLVM IR by executing explicitly the pipeline of LLVM passes we give as arguments - note that we do not call the *indvars* module because it changes the names of variables, which would disallow to later recover from LLVM IR to C/C++ the names from the C source file. Among those modules, we dynamically load the *SRA (Symbolic Range Analysis)* and our modified *LoopVectorizeOpincaa* passes. Next, we run the back end, *llc*, to generate CPU assembly instructions and Connex-S vector assembly code. We then replace the vectorized loops in the source C file with the associated obtained OPINCAA kernels and coordination code in order to obtain the final OPINCAA program, by using a simple tool. With this last step, we essentially perform a simple source-to-source transformation, which makes the code CPU independent. Generating C++ code allows using the OPINCAA C++ framework in order to write vector-length agnostic Connex-S assembler code, which performs CVL-parametric strip-mining by relying on the OPINCAA JIT assembler and reading at runtime the CVL program environment variable. The resulting

C++ OPINCAA program can be compiled with a standard tool like GCC, preferably at the maximum optimization level for the benefit of the host code.

Since an LLVM back end must have a scalar CPU processor, not only a vector unit, we create the Connex-S back end by adding the Connex-S vector instructions to the existing LLVM *eBPF (extended Berkeley Packet Filter)* [31] back end. We could start from a more common back end such as the one for ARM, but since our method discards the scalar assembly code at the end and replaces it with the original sequential C code, we allow ourselves to use something simpler. We add to it vector instructions getting inspired from the *MSA (MIPS SIMD Architecture)* vector instructions specified in the LLVM *Mips* back end.

Note that in the LLVM IR machine model, the virtual vector units are tightly coupled to the CPU, which runs the sequential LLVM code. On the other hand, Connex-S is normally an accelerator, loosely integrated with the CPU, with its own separate memory space, as already discussed, so the communication and synchronization between the two needs to be performed explicitly, using OPINCAA's coordination *Application Programming Interface (API)*. For this, we generate the calls for these memory transfers in our *LoopVectorizeOpincaa* module.

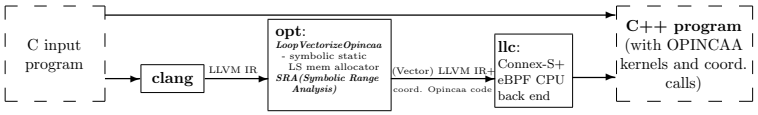

FIG. 2. The stages of the Connex-S OPINCAA LLVM compiler

We extend LLVM's standard *LoopVectorize* module, which automatically transforms, if profitable, sequential innermost loops of the input LLVM IR program into vector LLVM code, to implement most non-back end functionality:

- a symbolic static memory allocator for Connex-S's LS memory;
- generation of optimal, aggregated I/O transfer calls between the Connex-S LS memory and the system RAM, of OPINCAA kernel begin and end primitives, of kernel execution calls and the reads of the reduction results;
- the logic to create new appropriate vector reads and writes to properly address the LS memory instead of the system RAM;
- the generation of loop headers and footers inside the Connex-S vector kernel;
- to achieve a respectable performance we need to reduce the size of the kernel by generating for the innermost two levels of C loop nests an OPINCAA *REPEAT* loop containing a *host-side* C++ *for* loop (this normally results in a kernel with minimal number of vector instructions if all the original trip counts are at least CVL), while for simple C loops we generate only *REPEAT*. Note that for deeper loop nests we simply leave unchanged the outer loops.

5.1. Example of Compiled Code

When compiling the simple C program from Listing 1, implementing a sum-reduce pattern over an array of size variable N, we obtain the program in Listing 2 containing OPINCAA coordination and assembly code.

The *connexGlobal* C++ object encapsulates the accelerator functionality. Its *writeDataToConnexPartial()* method performs a blocking I/O transfer with the N short elements of array C from the CPU memory to the Connex-S LS memory, from offset 0. The OPINCAA Connex-S kernel gets assembled on the CPU when the OPINCAA program is running and starts being executed on the accelerator only when the CPU enters *executeKernel()*. *readReduction()* also is a blocking method, which waits in this case for the only reduction result from the kernel provided by the last vector instruction. Note that we use in

the OPINCAA Connex-S code a *symbolic* C/C++ operand for the number of iterations of the *REPEAT* instruction, which is to be translated to an immediate operand during the assembling, at runtime.

We also address the correctness of the generated Connex-S OPINCAA vector-length agnostic code in the case the trip count of the vectorized loop is not a multiple of CVL: we pad in method *writeDataToConnexPartial()* executed on the CPU array C with zero elements such that it occupies an integer number of vectors.

It is important to note again that this OPINCAA program performs CVL-parametric strip-mining and uses the *CVL* environment variable, which informs the program what is the number of lanes, s.t. it can run on a Connex-S accelerator with an arbitrary CVL.

```
// Assume N * sizeof(short) <=
//          CONNEX_MEM_SIZE
short SumReduce(short *C, int N) {
    short sum = 0;

    for (int i = 0; i < N; ++i)
        sum += C[i];

    return sum;
}
```

LISTING 1. C source program for array sum-reduction

```
int CONNEX_VL;
#define CVL CONNEX_VL /* for readability */
short SumReduce(short *C, int N) {
  short sum = 0;
  // If N % CVL != 0 we pad data with 0
  connexGlobal->writeDataToConnexPartial(
        C, N, 0 /* LS memory offset */ );
  BEGIN_KERNEL("SumReduce");
    EXECUTE_IN_ALL(
      R(0) = 0; R(1) = 1; R(2) = 0;
      R(3) = 0; /* accumulator */
      // CVL strip-mined dot product
      REPEAT((N / CVL) + ((N % CVL) > 0));
        R(4) = LS[R(2)];
        R(2) = R(2) + R(1);
        R(3) = R(3) + R(4);
      END_REPEAT;
      RED R(3);
    );
  END_KERNEL("SumReduce");
  connexGlobal->executeKernel("SumReduce");
  sum = connexGlobal->readReduction();
  return sum;
}
```

LISTING 2. The OPINCAA program generated by the Connex-S OPINCAA LLVM compiler from the code from Listing 1

6. **Experiments**

We now evaluate the performance of the code generated by our optimizing compiler.

For experiments, we use a Zedboard development platform with the Xilinx Zynq-7020 SoC, with an ArchLinux 1.4 distribution with Linux kernel 3.14.0, and GCC 8.2.0. Connex-S OPINCAA LLVM [2] is based on LLVM 8.0 from Mar 2019, with a *LoopVectorize* pass from LLVM 3.8, and the OPINCAA library is available for download at [3].

We refer the reader to our previous paper [16], which contains a few more experiments.

We present in Figure 3 the performance speedups of a few benchmarks written in C when running on a Connex-S machine with 128 lanes and an LS memory of 1024 lines synthesized on the Xilinx Zynq-7020 FPGA, clocked at 100 MHz w.r.t. the dual-core ARM Cortex A9 processor integrated into the Zynq SoC, at 667 MHz, equipped with an 8-stage superscalar pipeline with 128-bit NEON SIMD support - the board setup also is described by Bîră et al. [12]. Connex-S uses most of the resources of the FPGA. Note that on ARM, just as on Connex-S, we run the benchmarks compiled with maximum optimization level. Note that we do not perform loop tiling transformations on ARM since GCC offers little support for it and ARMv7's speculative hardware prefetcher is poor, not suitable for tiling.

All the benchmarks employ arrays with elements of native type 16-bit integer (*i16*), or of emulated types 32-bit integer (*i32*) and 16-bit floating-point (*f16*). The benchmarks perform: *matrix multiplication* (*MatMul*) for sizes 128×128, and 256×256, the second matrix being already transposed to allow better vectorization; *Sum of Squared Differences (SSD-1024)* and *Sum of Absolute Differences (SAD-1024)*, standard functions used in computer vision, compute statistics for all pairs of two groups of 64, 64 or 32 collections of 1024 i16, f16 or i32 elements, respectively. All these kernels, except *MatMul-128*, have input data of 256 KB, the size of the LS memory. Also, *MatMul-256.i32* requires tiling because its memory footprint is bigger than the 256 KB of the SPM.

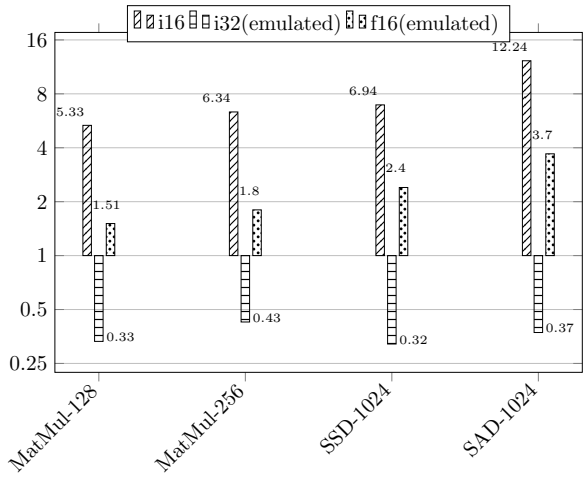

FIG. 3. Semi-log plot with speedups of the benchmarks on Connex-S with 128 lanes, at 100 MHz w.r.t. the dual-core ARM Cortex A9 at 667 MHz with two 128-bit NEON SIMD units

All i32 benchmarks achieve a subunitary speedup because of the big complexity of the i32 arithmetic operations emulated on Connex-S and because GCC vectorizes programs with i32 type for ARM. To be able to actually accelerate these i32 benchmarks on Connex-S, we should run them on a wider vector processor: for example, *SAD-1024.i32* on a Connex-S with 512 lanes should achieve a speedup factor of 1.4.

We experience a decent acceleration of the f16 benchmarks because ARMv7 does not support the f16 type natively either, so it has to convert it to f32 to perform native operations and then revert to f16, and these conversion operations have a big cost. A less important reason is the fact GCC 8.2 is unable to vectorize floating-point operations for ARM NEON.

We now measure the energy consumed by Connex-S and the Cortex A9 CPU integrated into the Zynq SoC when running our benchmarks. To measure the power consumption of Connex-S and the ARM CPU integrated into the Zynq SoC reported in Table 1 we use a daughtercard for Zedboard, which puts in contact the *Current Sense* and the *XADC Header* connectors of Zedboard, the latter being the interface to Xilinx's *Analog Mixed Signal (AMS)* module. We read the XADC registers from the Linux file system to obtain the instantaneous power consumption every 0.1 seconds. We can also sample ten times faster, which seems it is not too invasive and should increase precision. Then, we compute the energy consumption using the integration trapezoidal rule given the measured power values.

Breaking down the energy consumption of the ARM CPU from the FPGA on the Zynq-7020 SoC is somewhat difficult. Fortunately, we use the existing linear regressive model of the ARM Cortex A9 CPU integrated into the Zynq-7020 SoC from Wu et al. [50] to determine the idle power consumption of Cortex A9 with a standard Linux distribution, which in our case is 0.224 Watts. We also use the Xilinx Power Estimator model of the CPU to estimate the idle power consumption of the ARM processor with a load of 2% [51] and obtain a similar result.

We see in Table 1 that the energy consumption of Connex-S is better for tests with i16 type w.r.t. the ARM Cortex A9 given the fact we achieve high speedups, but worse for types i32 and f16. We note that the energy-saving ratio is almost constantly proportional to the execution time speedup with a factor of 0.3x. This indicates the average power consumption of Connex-S and the CPU is almost constant for all our benchmarks. We remind that our Connex-S accelerator being currently implemented in an FPGA consumes an order of magnitude more power than a dedicated IC such as the one reported in Maliţa et al. [33].

We present in Table 2 with the energy consumption of the Connex-S assembly instructions. Power consumption varies also with ambient temperature, but we are able to adjust the power of idle functioning of the Zynq-7020 SoC to perform useful measurements of our benchmarks on the CPU or the accelerator.

Table 1. The energy consumption in Joules of the benchmarks when executed on Connex-S with 128 lanes, clocked at 100 MHz, together with the CPU performing just I/O - see column *Connex-S + $CPU_{I/O}$*. Also the consumption of the same tasks running just on the dual-core ARM Cortex A9 at 667 MHz together with Linux OS - see column *CPU*. We also present the energy-saving ratio when running on Connex-S instead of CPU and compare it to the execution time speedup.

Benchmark	**Connex-S + $CPU_{I/O}$**	**CPU**	**Eng.-saving ratio**	**Ratio / speedup**
MatMul-128.i16	0.003115	0.004374	1.404x	0.263x
MatMul-256.i16	0.022218	0.039128	1.761x	0.278x
SSD-1024.i16	0.005047	0.012177	2.413x	0.348x
SAD-1024.i16	0.005572	0.020857	3.356x	0.274x
MatMul-128.i32	0.019287	0.002644	0.137x	0.415x
MatMul-256.i32	0.190326552	0.031266	0.163x	0.382x
SSD.i32	0.013613	0.001497	0.11x	0.344x
SAD.i32	0.014783	0.001742	0.118x	0.319x
MatMul-128.f16	0.148319	0.062082	0.419x	0.277x

We observe from Table 2 that: i) **nop** has the smallest energy consumption, for the obvious reason it does not perform any computation; ii) all arithmetic and logical instructions, except **multlo** and **multhi**, which only copy registers, consume a similar energy quantity, with an average of 11.41 nJ; iii) **vload** is more energy consuming than **ldix** because **vload**'s immediate operand value is transmitted from the controller to each lane, while **ldix** loads the value from the execution unit itself; iv) a bit surprisingly, **cellshl** and **cellshr** consume little energy because of the locality of the data movements in each cycle.

We have included the values from Table 2 in our OPINCAA architectural simulator, in order to provide a simple power model for our Connex-S processor to estimate the energy consumption of the accelerator when executing a vector kernel. It is easy to adapt our power model for a different Connex-S processor implementation such as a dedicated IC.

A widely used metric is the *performance per Watt*, which is the arithmetic instruction throughput of a processor divided by the average power consumption. We compute this metric for Connex-S by using the number of equivalent integer or floating-point *scalar* operations, divided by the execution time and power. For the *MatMul-128* benchmark, we achieve on Connex-S 1.12 GOPS/Watt for type i16, 0.172 GOPS/Watt for type i32, and 0.031 GFLOPS/Watt for type f16.

In Table 3 we show the energy consumption variation of the *MatMul-128.i16* kernel w.r.t. the *Connex-S width*, CVL. We note that increasing CVL leads to saving energy since we decrease the number of vector instructions required to execute the kernel and keep the power consumption of the instruction sequencer basically constant, which makes the energy efficiency of a vector instruction grow together with the CVL. This is why, for example, Connex-S with 128 lanes consumes 1.6, respectively 1.8 times less energy than Connex-S with CVL 32. This trend can also be noticed in Figure 14 of Waeijen et al. [47] for a vector processor similar to ours. Also, Inoue [25] notices that using the SIMD units on a multi-core x86 processor for an application performing sorting saves energy, even if it increases the average power.

Table 2. The average energy consumption in nanojoules when executing a vector assembly instruction or I/O primitive on Connex-S with 128 lanes, clocked at 100 MHz. All assembly instructions take 1 cycle to be processed. We provide random input operands to the vector instructions. For comparison, the energy consumption of an *add* or *sub* scalar instruction of the Cortex A9 host is 0.1 nJ.

Connex-S Instruction	**Avg. eng. consumption [nJ]**
add	12.1
sub	12.2
addc	12.2
subc	12.2
mult[.u16]	12.6
multlo	0.5
multhi	0.5
not	7.5
or	11.5
and	11.8
xor	12.5
shl	12.0
ishl	12.1
shr	12.0
ishr	12.0
ishra	12.4
popcnt	7.2
eq	10.5
lt	10.8
ult	10.8
iread	2.6
read	5.6
iwrite	9.9
write	12.8
ldix	0.5
vload	8.4
endwhere	0.1
whereeq	0.1
wherelt	0.1
wherecry	0.1
disablecells	1.7
enableallcells	0.1
setlc	0.15
ijmpnzdec	0.15
nop	0.1
cellshl	0.2
cellshr	0.2
red[.u16] with *readReduction()*, for 1 cycle	23
vector kernel transfer for 1 instruction	9.5
writeDataToConnex() for 1 cycle	3.8

Table 3. The energy consumption in Joules of the *MatMul-128.i16* benchmark for different Connex-S processors with 32, 64 and 128 lanes, clocked at 100 MHz. We also present the number of vector instructions executed for each experiment and the ratio of the energy saving w.r.t. the consumption for 32 lanes when CVL is 64 or 128.

CVL	**32**	**64**	**128**
Energy consumption	0.005708 [J]	0.003801 [J]	0.003115 [J]
num. vector instr.	557,950	328,574	213,886
ratio wrt CVL=32	1.0x	1.5x	1.8324x

7. **Conclusions**

This paper described how we efficiently compile sequential C programs to obtain *vector-length agnostic (VLA)* code for the wide Connex-S vector accelerator. The generated C++ program with host code and Connex-S assembler code uses the OPINCAA JIT vector assembler and coordination library, which can run the VLA code on accelerators of different width.

The Connex-S OPINCAA LLVM compiler generates code of good quality. We report speedups of up to 12.24 when using the Connex-S vector accelerator with 128 lanes clocked at a frequency of 100 MHz w.r.t. the dual-core ARM Cortex A9 running at 667 MHz. Also,

we achieve a modest energy efficiency improvement average of 1.1 times when accelerating on Connex-S the computation. However, we could save at least an order of magnitude more energy when using a reported prototype Connex-S dedicated integrated circuit also fabricated in 28nm technology [33].

Our most original contributions in the Connex-S OPINCAA LLVM compiler are: i) achieving VLA compilation in LLVM, which is something novel, basically only done for the ARM SVE ISA extension for both LLVM and GCC, and for the .Net IR in order to target arbitrary SIMD ISAs [40]; ii) the simple instruction set supports only 16-bit integer native arithmetic vector operations, which forces us to emulate efficiently other useful types like 16-bit floating-point and 32-bit integer using a combination of manual and simple compiler optimizations.

For a complete description of the compiler, more details on the Connex-S architecture and on the energy measurements we invite the reader to consult the author's PhD thesis.

Acknowledgments

We would like to thank professor Gheorghe M. Ştefan and Lucian Petrică, Ovidiu Plugariu, Radu Hobincu, Călin Bîră, Michael Kruse, Albert Cohen, Alex Zinenko, Tobias Grosser and Gleisson Mendonca for the help they offered during this project. Also thanks to the people from the *llvm-dev* mailing list: Hal Finkel, Daniel Sanders, Eli Friedman, Tom Stellard, Renato Golin, Quentin Colombet and so on.

REFERENCES

[1] Connex Accelerator Controller Specification, 2017.

[2] Connex OPINCAA LLVM compiler source repository, http://gitlab.dcae.pub.ro/research/ConnexRelated/OpincaaLLVM.

[3] The Connex OPINCAA library, http://gitlab.dcae.pub.ro/research/opincaa.

[4] S. G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall, 1989.

[5] R. Allen and K. Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *TOPLAS'87*.

[6] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, and M. Lam. The Warp Computer: Architecture, Implementation, and Performance. *IEEE T.C. 1987*.

[7] ARM Manchester Design Center. Support for Scalable Vector Architectures in LLVM IR, 2016.

[8] A. Armejach et al. Stencil codes on a vector length agnostic architecture. PACT'18.

[9] K. Asanovic. *Vector Microprocessors*. PhD thesis, 1998.

[10] K. Asanović and R. Espasa. The RISC-V Vector ISA, 7th RISC-V Workshop, 2017.

[11] D. F. Bacon et al. Compiler Transformations for High-performance Computing. *ACM Comput. Surv. 1994*.

[12] C. Bîră, R. Hobincu, et al. Energy-Efficient Computation of L1 and L2 Norms on a FPGA SIMD Accelerator, with Applications to Visual Search. In *CSCC'14*.

[13] C. Bîră, L. Petrică, and R. Hobincu. OPINCAA: A Lightweight and Flexible Programming Environment For Parallel SIMD Accelerators. *RJIST'13*.

[14] G. E. Blelloch. *Vector Models for Data-parallel Computing*. MIT Press, 1990.

[15] D. Brooks and M. Martonosi. Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance. HPCA'99.

[16] A. Şuşu. Compiling Efficiently with Arithmetic Emulation for the Custom-Width Connex Vector Processor. WPMVP'19.

[17] G. De Micheli et al., editors. *Readings in Hardware/Software Co-design*. 2002.

[18] J. Dongarra et al. *Sourcebook of Parallel Computing*. Morgan Kaufmann 2003.

[19] A. Eichenberger et al. Optimizing Compiler for the CELL Processor. PACT'05.

[20] Francesco Petrogalli. A Sneak Peek into SVE and VLA Programming, ARM White Paper, 2016.

[21] Gheorghe M. Ştefan. The Connex Instruction Set Architecture, 2015.

[22] T. Grosser and T. Hoefler. Polly-ACC Transparent Compilation to Heterogeneous Hardware. ICS'16.

[23] A. Hennequin et al. Designing Efficient SIMD Algorithms for Direct Connected Component Labeling. WPMVP'19.

[24] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. 2017.

[25] H. Inoue. How SIMD width affects energy efficiency: A case study on sorting. In *CoolChips'16*.

[26] K. Karuri, R. Leupers, G. Ascheid, et al. Design and Implementation of a Modular and Portable IEEE 754 Compliant Floating-point Unit. DATE'06.

[27] O. Krzikalla et al. Scout: A Source-to-source Transformator for SIMD-Optimizations. Euro-Par'11.

[28] I. Kuon and J. Rose. Measuring the Gap between FPGAs and ASICs. FPGA '06, 2006.

[29] S. Larsen. Increasing and Detecting Memory Address Congruence. PACT'02.

[30] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. CGO'04.

[31] Linux Kernel. https://www.kernel.org/doc/Documentation/networking/filter.txt, 2014.

[32] K. Mai et al. Smart Memories: A Modular Reconfigurable Architecture. ISCA'00.

[33] M. Maliţa and G. M. Ştefan. Map-Scan Node Accelerator for Big-Data. In *2017 IEEE Big Data*.

[34] M. Malita et al. Complex vs. Intensive in Parallel Computation. In *ICCGI'06*.

[35] G. Mendonça et al. DawnCC: Automatic Annotation for Data Parallelism and Offloading. *TACO'17*.

[36] S. Mittal. A Survey of Techniques for Architecting and Managing GPU Register File. *IEEE TPDS'17*.

[37] A. Munshi et al. *OpenCL Programming Guide*. 2011.

[38] D. Naishlos. Autovectorization in GCC. Proceedings of the 2004 GCC Developers Summit.

[39] D. Nuzman et al. Multi-platform Auto-vectorization. CGO'06.

[40] D. Nuzman et al. Vapor SIMD: Auto-vectorize Once, Run Everywhere. CGO'11.

[41] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. 2013.

[42] L. Petrică et al. VASILE: A reconfigurable vector architecture for instruction level frequency scaling. FTFC'13.

[43] R. G. Scarborough and H. G. Kolsky. A Vectorizing Fortran Compiler. *IBM J. Res. Dev.*, 1986.

[44] D. B. Skillicorn et al. Models and Languages for Parallel Computation. *ACM Comput. Surv.'98*.

[45] G. M. Ştefan and M. Maliţa. Can One-Chip Parallel Computing Be Liberated From Ad Hoc Solutions? In *CSCC'14*.

[46] N. Stephens, S. Biles, J. Eapen, et al. The ARM Scalable Vector Extension. *IEEE Micro'17*.

[47] L. Waeijen, D. She, H. Corporaal, and Y. He. A Low-Energy Wide SIMD Architecture with Explicit Datapath. *JSPS'15*.

[48] A. Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, 2016.

[49] A. Waterman et al. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0. Technical report, 2014.

[50] Y. Wu, J. Nunez-Yanez, R. Woods, and D. S. Nikolopoulos. Power Modelling and Capping for Heterogeneous ARM/FPGA SoCs. In *FPT'14*.

[51] Xilinx. Xilinx Power Estimator tool, http:// www.xilinx.com/products/technology/power/xpe.html.