VECTOR-LENGTH AGNOSTIC COMPILATION FOR THE CONNEX-S EMBEDDED WIDE VECTOR ACCELERATOR

Alexandru E. Şuşu[1]

These days we experience a fertile moment for parallel computing, mostly due to advances in compiler technology, the maturation of mathematical formalisms for it and the popularization of hardware design.

The major contribution of the thesis is the compilation of sequential C programs for Connex-S, a competitive, scalable and customizable, wide vector accelerator for intensive embedded applications with 32 to 4096 16-bit integer lanes and a limited capacity local scratchpad memory. This contribution is well-evidenced in our recent paper Şuşu 2020. The Connex-S processor is designed in our *D.C.A.E.* (in Romanian: *Dispozitive, Circuite şi Arhitecturi Electronice*, meaning *Devices, Circuits and Electronic Architectures*) laboratory in the *E.T.T.I.* (in Romanian: *Electronică, Telecomunicaţii şi Tehnologia Informaţiei*, meaning *Electronics, Telecommunications and Information Technology*) department at the Politehnica University of Bucharest proposes a promising research an embedded wide vector architecture called Connex-S. The Connex-S accelerator draws its efficiency from delegating the compiler to extract parallelism by vectorizing scalar code, and to perform explicit communication in an optimized fashion between the lanes of the processor. Obviously in software parallelism extraction is considerably better than when it is performed by hardware as it is for example the case for superscalar processors. Connex-S is a low-power processor. The reason is that *Single Instruction Multiple Data (SIMD)* array or vector architectures, besides being a scalable way to design parallel architectures C. E. Kozyrakis et al. 2003, they also achieve better energy efficiency because (i) SIMD (array) processors use considerably fewer resources for the control unit than for the datapath Waeijen et al. 2015; Şuşu 2019; Patterson et al. 2017, (ii) we delegate the compiler to extract *Instruction Level Parallelism (ILP)* instead of the hardware as it is the case for superscalar processors, and (iii) we can maintain the computational throughput by lowering the frequency and the supply voltage since we have an increased level of hardware parallelism Chandrakasan et al. 1992. C. Kozyrakis 2002 states that in a SIMD (array) processor increasing the number of lanes allows for a decrease in power consumption without changing the peak computational throughput of the system.

[1]Ph.D. student, ETTI, UPB (Politehnica University of Bucharest), Romania, e-mail: alex.susu@gmail.com. This document is a short version of his Ph.D. thesis.

The number of lanes of the SIMD processor is called *width* or *vector length*.

The Connex-S ISA is described in the PhD thesis. But a complete technical document for the Connex-S ISA is Ştefan 2015.

Any digital processor requires good software development tools to be embraced by the community. These tools are the assembler, the compiler for C or C++, the runtime library, and additional software libraries and frameworks. Ideally, the runtime library contains few elements written manually in assembler by expert programmers such as floating-point emulation or special *Operating System (OS)* calls. On the other hand, a C compiler includes all the complex logic to translate general C code to assembly code. While this is something easy to do nowadays for scalar processors, auto-vectorization, the compilation for vector processors remains, even after almost 40 years of research, a difficult task. For example, a lot of non-regular code such as code with complex data structures, recursive code, or with parallel patterns such as reduction or scan remains difficult to be auto-vectorized.

Therefore, as a doctoral student, I had to create a decent Connex-S vector runtime library while developing our Connex-S compiler. The Connex-S compiler uses the state-of-the-art LLVM compiler Lattner et al. 2004, but since the LLVM compiler itself does not include support for vectorization for all possible parallel-patterns such as the scan operator I had to conservatively create manually optimized vector assembly functions in our runtime library for them and call them explicitly in our code. This practice is good for now, but, again, we hope in the next years to be able to recognize such patterns in our compiler and generate efficient code for them.

The **targets of my research** were from the beginning: (i) to create an optimizing compiler, as general as possible for the Connex-S vector processor, together with a good supporting runtime library; (ii) to propose power and energy optimizations as much as possible in the compiler. I was able to address energy optimizations only as part of the runtime library, but not in the compiler because the processor power management infrastructure was difficult to build and there are few, complex energy-saving optimizations that can be addressed by the compiler.

Since Connex-S is an accelerator loosely integrated with the host it uses a programming model similar to the OpenCL language called OPINCAA Bîră et al. 2013. Therefore, our compiler toolchain uses the LLVM framework and targets OPINCAA, which is more exactly a *Just-In Time (JIT)* vector assembler and coordination C++ library for Connex-S accelerating computations for an arbitrary CPU. Therefore, we address in the compiler middle end aspects of efficient vectorization, communication, and synchronization. We perform quantitative static analysis of the program useful, among

others, for the symbolic-size compiler memory allocator and the coordination mechanism of OPINCAA. By using JIT vector assembling and by encoding the vector length of Connex-S as a parameter in the generated OPINCAA program, we achieve vector-length agnosticism to support execution on distinct embedded devices, such as several digital cameras with different resolutions, each equipped with custom-width Connex-S accelerators meant to save energy for the image processing kernels.

We encourage using a *Vector-Length Agnostic (VLA)* compiler, mainly because we should design an embedded system with one or more *custom-width* Connex-S processors adapted to the frequent program input sizes to reduce the energy consumption when accelerating any of the following benchmarks: (i) *Basic Linear Algebra Subprograms (BLAS)* such as matrix multiplication, (ii) *Computer Vision (CV)* transformations or (iii) *Deep Neural Networks* and other intensive machine learning tasks. The experiments in Figure 14 from Waeijen et al. 2015 suggest that to save energy, we should accelerate this type of regular code kernels with wide Connex-S processors with a vector length closest to the trip count of the vectorized loop. Therefore, in case we manufacture several digital cameras with different factory established (maximum) resolutions performing various image processing computations such as image noise removal by using a CV convolution transformation on the captured frames of the corresponding factory size, to save energy we need to employ Connex-S accelerators with vector lengths close to the frame width.

1. **Detailed Description of the Connex-S Compiler**

The most challenging steps the compiler needs to perform are data tiling, loop vectorization Naishlos 2004; Nuzman et al. 2006 with symbolic quantitative static analysis, and efficient non-native arithmetic and logical operation emulation in the back end.

We present in Figure 1 a flowchart with the stages of our compiler toolchain together with details on how we can deploy the resulting C++ program to various embedded platforms with custom-width Connex-S accelerators. We first determine for our C code what is the size of the vector kernel compiled from the original C program through simple OPINCAA JIT assembling, the vector kernel memory footprint, and the loop nest trip counts. We obtain the memory footprint through static analysis with the LLVM *Symbolic Range Analysis (SRA)* pass Mendonça et al. 2017. If the footprint of our program is a constant smaller than the *Connex-S scratchpad memory size*, normally 256 KB, then we simply provide the original C program to the Connex-S OPINCAA LLVM compiler and we are done. Otherwise, we require first to apply loop tiling, a standard transformation splitting a loop in two nested loops, an exterior one called tile loop and an internal one called element (point) loop, with every element loop processing a smaller interval of iterations than the original one Kennedy

et al. 2002. In such a case, our next step is to perform optimal data tile size selection using our Connex-S accelerator cost model such that we minimize the total kernel execution time subject to fitting larger program data in the *Local Storage (LS)* memory. Then we employ PPCG, which takes as input the given source C program and the optimal tile sizes. PPCG applies polyhedral modeling *The Polyhedral Model, http://polyhedral.info* 2020 and data dependence analysis to check whether the requested loop tiling is legal, in which case it generates the transformed C program, which we give as input to the Connex-S OPINCAA LLVM compiler.

We now present the steps performed in Figure 1 by the OPINCAA LLVM compiler. We parse the (transformed) C source file with the *clang* command, the front end, which generates unoptimized LLVM IR code. The *opt* command then optimizes the LLVM *Intermediate Representation (IR)* by executing the pipeline of middle-end LLVM 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 generated OPINCAA kernels and coordination code to obtain the final OPINCAA program, by using a simple tool we wrote in C++, *ReplaceLoopsWithOpincaaKernels*. With this last step, we can consider we perform a simple source-to-source transformation, which makes the code CPU independent. We generate C++ code using the OPINCAA C++ framework. This allows the vector-length agnostic Connex-S assembler code with CVL-parametric strip-mining to rely on the OPINCAA JIT assembler, which also reads at runtime the concrete value of 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)* back end. We could start from a more common back end such as the one for ARM, but since our method discards the CPU assembly code at the end and replaces it with the non-vectorized original C code, we allow ourselves to use something simpler. eBPF is a simple 64-bit RISC processor The Linux Kernel Archives 2014, an extension of the BPF architecture McCanne et al. 1993, normally interpreted by the Linux (or other Unixes) kernel and has the smallest LLVM back-end implementation in the repository. We add to it vector instructions getting inspired from the *MSA (MIPS SIMD Architecture)* vector instructions specified in the LLVM *Mips* back end.

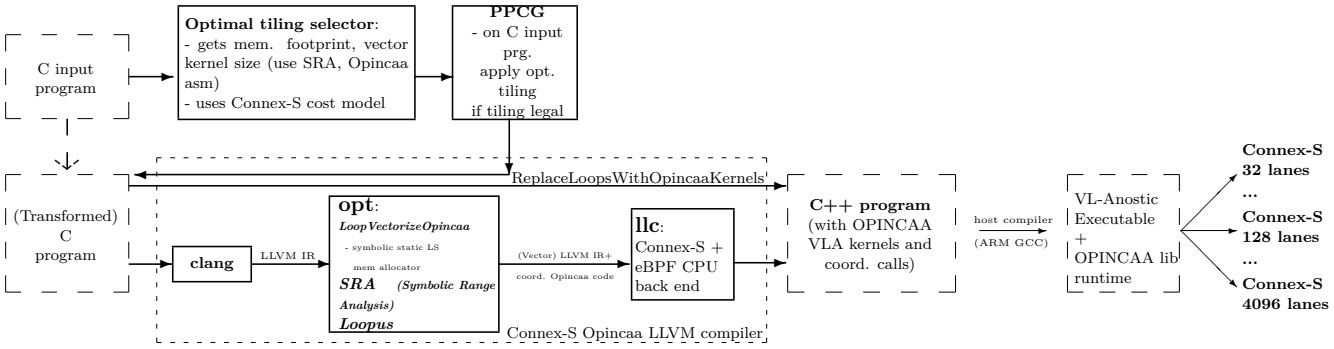

FIG. 1. The stages of the Connex-S compiler toolchain. Also, at the right end, we present examples of deployment to embedded systems with custom-width Connex-S accelerators with a number of lanes between 32 and 4,096.

Note that in the LLVM IR machine model, the vector unit is tightly-coupled to the CPU, which runs the sequential LLVM code. However, Connex-S is an accelerator, loosely integrated with the CPU, with its own separate memory space, so we need to explicitly perform the communication and synchronization between the two, using OPINCAA's coordination *Application Programming Interface (API)*. Therefore, we generate the OPINCAA coordination calls for these the OPINCAA memory transfers for memory and reduction transfers, execution and synchronization in our *LoopVectorizeOpincaa* module.

We extend LLVM's standard *LoopVectorize* module, which automatically transforms, if feasible and profitable, sequential innermost loops of the input LLVM IR program into vector LLVM IR code, to implement most (note that we add in the middle-end also the SRA and Loopus passes) new middle-end functionality:

- (i) a *symbolic-size* static memory allocator for the *Local Storage (LS)* memory;
- (ii) generation of optimal, aggregated I/O transfers between the system RAM, and the Connex-S LS memory (using the methods *writeDataToConnexPartial()*, *readDataFromConnexPartial()*), of OPINCAA *kernel begin* and *end* primitives, of kernel execution calls and the reads of the reduction results;
- (iii) the logic to create new appropriate vector reads and writes properly addressing the LS memory instead of the system RAM;
- (iv) the generation of loop headers and footers inside the Connex-S vector kernel. We have two alternatives to implement loops in the vector kernels: using host-side C++ *for* loops in OPINCAA, which actually completely unroll the body of Connex-S instructions or using Connex's hardware loop mechanism.
- (v) to get the start and end lines and columns in the C source file of the loops being vectorized in order to replace them later with OPINCAA code. For this, we use the *ScopeTree* class from the *DawnCC* compiler distribution Mendonça et al. 2017.

- (vi) for better performance we also need to reduce the size of the kernel by generating for the innermost two levels of C loop nests normally 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*. This transformation works if the second innermost loop has basically no code besides the inner loop aside from e.g. initialization of reduction variables, which means we can handle only perfectly nested loops Kodukula et al. 1997 with our transformation. Note that for deeper loop nests we simply leave unchanged the outer loops.

Note that the vectorization of the innermost loop generates a *vector.body* basic block, but also prologue in the *vector.ph* block and epilogue vector code in *middle.block*, in which we put the last part of the innermost loop, either an OPINCAA *END_REPEAT* instruction or an inline assembly pseudo-instruction to end the host-side C++ for loop. We simply collect all this vector code with the *ReplaceLoopsWithOpincaaKernels* tool and put it in an OPINCAA kernel. Also, as discussed previously, for efficiency we generate a compact loop for the second innermost loop if it has no code besides the inner loop by transforming it to a Connex-S REPEAT loop.

Control divergence due to if-then-else C branches is handled by the *LoopVectorizeOpincaa* module, by performing *if conversion*, which means the control flow of the branches is replaced with *select* LLVM vector IR instructions. However, if the structure of the loop to be vectorized is complex, it is possible that if conversion and vectorization fail in LLVM.

2. Automating the Specification of Lowering Efficiently Non-native Arithmetic and Logical Operations

Due to the lack of space, we briefly present the most interesting results of the semi-automatic methodology we employ for the efficient emulation of the non-native LLVM IR operations for types such as 32-bit integer or 16-bit floating point using our Connex-S OPINCAA LLVM compiler. We refer the reader to our Section 5.3 of the PhD thesis describing in more detail the aspects of this section.

The method we propose consists of: i) manually creating optimized OPINCAA vector assembly kernels we call *code templates* for the emulation of each of these non-native operations; ii) generating automatically C++ API code for LLVM's instruction selection pass, which lowers a non-native instruction to its respective predefined *code template*. This achieves efficient emulation of non-native operations in the sense that the LLVM compiler generates inlined emulation code for a program with non-native arithmetic operations, optimized by all the standard passes like register allocation

and *Common Subexpression Elimination (CSE)* basically starting from the manually optimized code templates.

For the method to work well we also devise a code generator tool part of the OPINCAA library with a simple algorithm to automate the specification of LLVM C++ API lowering code in the instruction selection pass, since manually writing C++ code can be error prone.

3. **Optimal Data Tile Size Selection During Compilation**

Since Connex-S has a limited capacity local scratchpad memory of 256 KB normally, we present how we also use the PPCG C-to-C code generator to perform data tiling to minimize the total kernel execution time, subject to fitting larger program data in the local memory. To compute an optimal tiling, we first introduce the accurate cost model of our Connex-S embedded accelerator, currently synthesized in a Xilinx Zynq-7020 FPGA, to determine the total execution time of a tiled routine. We run at compile time this optimization problem whether the C source program does not have parametric size arrays for which we cannot infer transfer costs statically. Our work resembles the one of Lin et al. 2010 that build a similar by principle cost model to achieve optimal tiling for the IBM Cell processor.

Connex-S has an accurate and simple to calculate cost model if few stalls occur in the system, which happens for our kernels that get completely vectorized and for which we read efficiently the reduction results, in blocks of at least a few KB. This is of no surprise since Connex-S is a wide vector accelerator, with a software-managed *scratchpad memory (SPM)* instead of caches, which makes vector execution predictable, with no speculative or out-of-order execution, and we implement Connex-S inside a Xilinx Zynq-7020 FPGA on Zedboard, a rather simple embedded prototype platform. We also experience some stalling behavior when the instruction or the reduction FIFO becomes full, which is under 5% of the total kernel execution time for non-degenerate tilings. Therefore, we perform rigorously measurements in many different conditions on the Zedboard and fit our cost model to have good accuracy.

```
typedef int16_t Type;

#define N 512
Type A[N][N];
Type B[N][N];
Type C[N][N];

void MatMul_BTransposed() {
  int i, j, k;

  for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j) {
      C[i][j] = 0;
      for (k = 0; k < N; ++k)
          C[i][j] += A[i][k] *
                     B[j][k];
    }
}
```

LISTING 1. C source program for matrix multiplication

We provide an example of compilation where it is necessary to perform loop tiling to be able to fit the data accessed in the Connex-S LS memory of 256 KB. When compiling with the PPCG tool the simple C function from Listing 1, which we refer to as the *MatMulBT-512* benchmark, implementing the multiplication of two square matrices of 512×512 elements, with elements of type *int16_t*, stored in row-major order, with the second operand transposed to allow vectorization, we obtain the program in Listing 2.

```
int16_t Atile[182][512];
int16_t Btile[74][512];
int16_t Ctile[182][74];

void MatMul_BTransposed() {
  #define min(x, y) (x < y ? x : y)

 #ifdef NOT_FOR_CONNEX_LLVM_COMPILER
  // The i tile loop
  for (int it = 0; it < 512; it += 182) {
    int callWriteDataToConnexForFirstArray = 1;
     // For data reuse in the OPINCAA C++ program

    // Copying data from A into Atile
    for (int ip = 0; ip < min(182, 512-it); ip++)
      for (int k = 0; k < 512; k++)
        Atile[ip][k] = A[it + ip][k];

    // The j tile loop
    for (int jt = 0; jt < 512; jt += 74) {
      // Copying data from B into Btile
      for (int jp = 0; jp < min(74, 512-jt); jp++)
        for (int k = 0; k < 512; k++)
          Btile[jp][k] = B[jt + jp][k];
 #endif
  // Continued on right column
```

```
  // Continued from left column

      //Only this code is vectorized on Connex-S
      for (int ip = 0; ip < min(182, 512-it);
                          ip++) // i point loop
        for (int jp = 0; jp < min(74, 512-jt);
                         jp++) { // j point loop
          Ctile[ip][jp] = 0;
          for (int k = 0; k < 512; k++)
            Ctile[ip][jp] += Atile[ip][k] *
                             Btile[jp][k];
        }

 #ifdef NOT_FOR_CONNEX_LLVM_COMPILER
      int counter = 0;
      // Putting back data from Ctile into C
      for (int ip = 0; ip < min(182, 512-it);
                                    ip++)
        for (int jp=0; jp < min(74, 512-jt);
                                    jp++)
          C[it + ip][jt + jp] = Ctile[ip][jp];

    } // End of jt tile loop
  } // End of it tile loop
 #endif
}
```

LISTING 2. C program generated by PPCG from Listing 1, simplified for readability, for tile size vector (182, 74, 512), for a 256 KB Connex-S LS memory

```
int CONNEX_VL;
...
void MatMul_BTransposed() {
  // Assuming: CVL in {32, 64, 128, 256, 512}
  #define min(x, y) (x < y ? x : y)

  // The i tile loop
  for (int it = 0; it < 512; it += 182) {
    int callWriteDataToConnexForFirstArray = 1;

    // Copying data from A into Atile
    for (int ip = 0; ip < min(182, 512-it); ip++)
      for (int k = 0; k < 512; k++)
        Atile[ip][k] = A[it + ip][k];

    // The j tile loop
    for (int jt = 0; jt < 512; jt += 74) {
      // Copying data from B into Btile
      for (int jp = 0; jp < min(74, 512-jt); jp++)
        for (int k = 0; k < 512; k++)
          Btile[jp][k] = B[jt + jp][k];

  // Performing data reuse for block Atile
  if (callWriteDataToConnexForFirstArray == 1) {
    // connexGlobal C++ object encapsulates
    //      accelerator functionality
    connexGlobal->writeDataToConnexPartial(Atile,
                          /* num elems written */
                          min(182, 512-it) * 512,
                        /* LS memory offset */ 0);
    callWriteDataToConnexForFirstArray = 0;
  }
  connexGlobal->writeDataToConnexPartial(Btile,
                       /* num elems written */
                         min(74, 512-jt) * 512,
                  /* LS mem offset */
                  min(182, 512-it) * 512 / CVL);

  // The OPINCAA vector kernel starts here
  BEGIN_KERNEL("allowRedefine_MatMul_BTransposed");
   EXECUTE_IN_ALL(
    R(0) = 0;
    R(1) = 1;

    // The i point loop
    for (int ip = 0; ip < min(182, 512 - it);
                                    ip++) {
 // Continued on right column
```

```
      R(2) = min(182, 512-it) * 512 / CVL;
                    // Btile start offset

      // Translation of j point loop jp
      REPEAT(min(74, 512-jt));
       R(3) = (ip * 512) / CVL;
                // vload index for Atile[ip]
       R(4) = 0; // accumulator

       // Vectorized innermost loop k
       //          (CVL strip-mined dot product)
       for (int kStripmine = 0; kStripmine < 512;
                              kStripmine += CVL) {
                  // Read Atile[ip][*] vector
         R(5) = LS[R(3)];
         R(3) += R(1);
                  // Read Btile[jp][*] vector
         R(6) = LS[R(2)];
         R(2) += R(1);
         R(6) * R(5); R(5) = MULTLO();
         // Accumulate (for dot prod)
         R(4) += R(5);
       }
       RED R(4); // compute Ctile[ip][jp]
      // End of translation of j point loop jp
      END_REPEAT;
    } // end of i point loop ip
   );
 END_KERNEL("allowRedefine_MatMul_BTransposed");

 connexGlobal->executeKernel(
             "allowRedefine_MatMul_BTransposed");
 connexGlobal->readCorrectReductionResults(Ctile,
              min(182, 512-it) * min(74, 512-jt),
                                sizeof(int16_t));

      // Putting back data from Ctile into C
      int counter = 0;
      for (int ip = 0; ip < min(182, 512-it);
                                      ip++)
        for (int jp = 0; jp < min(74, 512-jt);
                                      jp++)
          C[it+ip][jt+jp] =
                    *(&Ctile[0][0] + (counter++));

    } // end of j tile loop jt
  } // end of i tile loop it
}
```

LISTING 3. Simplified excerpt of the OPINCAA program generated by the Connex-S LLVM compiler from Listing 2

This C program from Listing 2, with loops tiled optimally s.t. the data fits correctly in the Connex-S LS memory and the program achieves the smallest execution time is then compiled with

the Connex-S OPINCAA LLVM compiler. We present in Listing 3 the final program containing OPINCAA coordination and vector assembly code.

Note that in Listing 1 we do not express the size of the matrix encoded as a C variable because it makes the problem tractable at compile time.

We obtain from the *SRA* pass invoked in our *LoopVectorizeOpincaa* LLVM module, once we have an idea what are the vector and coordination operations for our OPINCAA program, that the memory footprint of the loop nest in this function is 1,048,576 bytes. This indeed is the size of the two input matrices A and B. The reason we do not also include here the result matrix C is that Connex-S employs for efficiency the sum-reduction hardware functional unit to execute the vectorized innermost loop performing dot product, and the reduction result is sent directly to the CPU without being stored in the LS memory.

Our simple brute-force search algorithm returns an optimal performance tile size vector of (182, 74, 512). In this vector, we associate the sizes from left to right: for the outermost loop i with a tile size of 182 to the innermost loop k with a tile of 512 elements. In this case, the last size implies that we do not perform any tiling on loop k.

PPCG generates auxiliary arrays *Atile*, *Btile* to completely fit in 256 KB data from the original arrays A and B, respectively, before the actual multiplication code. Then, our Connex-S LLVM compiler generates after the input data copies into tiles and before executing the kernel the blocking *writeDataToConnexPartial()* I/O transfers from these tile arrays to Connex-S with an amount of data computed with the *SRA* pass—see Listing 3. Similarly, the *Ctile* array is copied at the end of the *jt for* loop to the right locations of the result matrix C, which is orchestrated with the fact LLVM adds later a *readCorrectReductionResults()* call immediately before this tile copy. Interestingly, PPCG places the copies from matrix A to array *Atile* outside the *jt for* loop, which achieves optimal data reuse in the sense that the same data block *Atile* is being reused for all possible values of *Btile*.

We explain now a few notations in the OPINCAA kernel in Listing 3: $R(0)$ refers to the Connex-S vector register 0, while $LS[R(3)]$ denotes for each lane the value stored at the address in the LS memory indicated by the respective element of $R(3)$. The two C++ *for* loops in the assembly vector kernel, *ip* and *kStripmine*, allow OPINCAA to completely unroll at runtime the assembly code contained in their body. The OPINCAA Connex-S kernel name has the *allowRedefine* prefix since we redefine the kernel and assemble it on the CPU for 21 times during the program execution. The kernel starts being executed on the accelerator only when the CPU enters *executeKernel()*. Then, the blocking *readCorrectReductionResults()* method waits for $min(182, 512 - it) \cdot min(74, 512 - jt)$ results from the reduction unit of Connex-S, which we use to update the corresponding elements of

matrix *Ctile*. This number of reduction results is computed in this example, as in other cases, by multiplying the trip counts of the two outermost point loops, but we plan to use in the near future the *Loopus* tool to support more general cases. Note that we do not store *Ctile* in the LS memory, because we compute every element of the array with the Connex-S sum-reduction hardware unit and sent directly to the CPU in the right order.

We stress that the assembly kernel is vector-length agnostic since we perform JIT assembling and CVL-parametric strip-mining with the *kStripmine for* loop, which completely unrolls the code inside its body. We can see the OPINCAA Connex-S vector code uses the C/C++ expression $(ip * 512)/CVL$ as a *symbolic* scalar immediate operand assigned to $R(3)$, which the assembler translates at runtime to a concrete value.

Note that the tiled C code from Listing 2 has some problems to compile with LLVM because of the *min* operators, being presented as such for readability. The reason is both the *SRA* algorithm and the OPINCAA kernel caching do not support *min* as a loop bound. More exactly: (i) we can consult this limitation of the *SRA* algorithm in Nazaré et al. 2014; and (ii) if the OPINCAA kernel contains *min* then caching the kernel binary stream at the first execution is unsound since the value returned by the *min* operator can change in a subsequent run, resulting in different kernel instructions than the ones cached. To address this issue we perform with PPCG a sort of loop unswitching transformation to the code in Listing 2, in which we duplicate the i point (ip) loop, each copy receiving one operand of the *min* operator as loop upper bound and being executed conditionally based on the value of the predicate $182 < 512 - it$. When compiling with the Connex-S LLVM compiler, each new ip loop becomes part of a new OPINCAA tile kernel with its own I/O calls. Similarly, we apply this transformation for the *jp REPEAT* loop. We do not present these corrected versions of the tiled C program or the OPINCAA C++ code due to lack of space, which also implies that Listing 3 does not perform kernel caching.

We also recommend consulting the simpler example of compilation without tiling of the program *SumReduce* from Şuşu 2019.

4. **Experiments**

We now present some meaningful benchmarks used, for example, in high performance and computer vision embedded applications, which our C optimizing compiler handles and evaluate the performance of the code generated for the Connex-S accelerator.

For experiments, we use a Zedboard development platform with the Xilinx Zynq-7020 *SoC (System on a Chip)*, with an ArchLinux 1.4 distribution with Linux kernel 3.14.0, and GCC 8.3.0

for ARM. The Connex-S compiler toolchain extends LLVM 8.0 from March 2019 together with a *Loop Vectorize* pass from LLVM 3.8 from Jul 2016, and PPCG with version 0.08.2. The OPINCAA library is available for download at the link in the reference DCAE 2019.

We present in Figure 2 the performance speedups of a few benchmarks written in C when running on a Connex-S accelerator with 128 lanes and a local SPM memory of 256 KB for program data, 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. While for Connex-S we use our OPINCAA LLVM compiler, for Cortex A9 we employ ARM GCC 8.3.0, and for both toolchains, we specify the maximum optimization level.

All the benchmarks employ arrays with elements of native type *i16*, but also of emulated types *i32* and *f16*, where appropriate. The benchmarks perform: *dot product (DotProd)* on arrays of 64K *i16* or *f16* elements, or 32K *i32* elements; *sum-reduction of population counts of words (CtPop-Reduce)* on an array of 128K elements of type *i16*—we do not run it also on *i32*, since it is an emulated type, less efficient on Connex-S; *matrix multiplication (MatMulBT-128/170/256/512/1024)* for operands of sizes 128×128, 170×170, 256×256, 512×512 and 1024×1024, the second matrix being already transposed to allow vectorization; *MatMul-128*, which also performs matrix transposition coded manually in vector assembler instead of the C language on the second input operand before the computation previously described for *MatMulBT-128* to follow the BLAS standard; *Sum of Squared Differences (SSD)* and *Sum of Absolute Differences (SAD)*, standard functions used for example in computer vision for the SIFT feature detection technique or for motion estimation Bocchino et al. 2006, compute statistics for all pairs of two groups of 2048 collections of 128 elements of type *i16*, *i32*, or *f16*; *covariance-128* and *correlation-128* are two benchmarks from the PolyBench suite Pouchet 2014 computing the covariance and correlation of 128 data points, each with 128 attributes of type *f16*; *filter2D-1024* performs a CV convolution transformation on a frame of $1,024 \times 768$ pixels, with a correlation kernel of size 15×15, being used to remove noise from images or to detect image features or edges.

The kernels *DotProd*, *CtPop-Reduce*, *MatMulBT-256.i16/f16* have input data of exactly 256 KB, the size of the SPM memory, while *MatMulBT-128/170*, *MatMul-128*, *covariance-128*, and *correlation-128* have less. However, the kernels *MatMulBT-256.i32*, *MatMulBT-512/1024*, *SSD-128*, *SAD-128*, and *filter2D-1024* have a memory footprint larger than the SPM, so we tile them optimally. We do not have loop tiling transformations for the programs running directly on ARMv7, since ARM

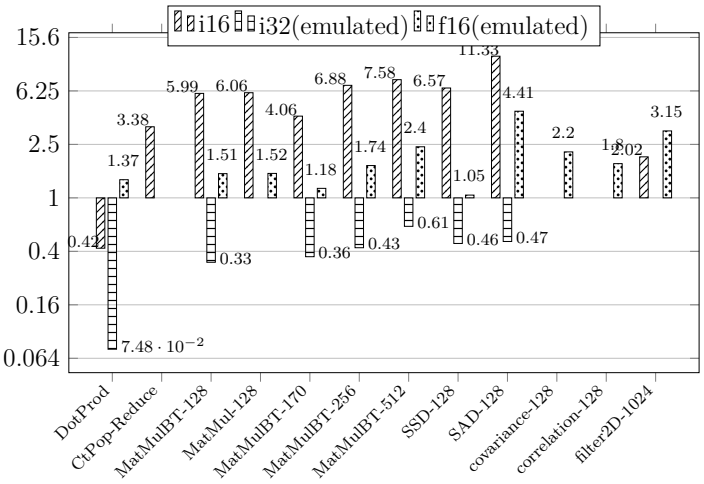

FIG. 2. Semi-log plot with the speedups of the benchmarks on Connex-S with 128 lanes, at 100 MHz w.r.t. the dual-core ARM Cortex A9 at 667 MHz with a total of two 128-bit Neon SIMD units.

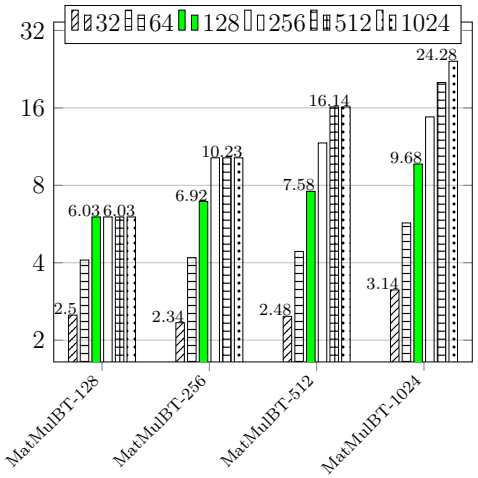

FIG. 3. Semi-log plot with the speedups of *i16* benchmarks on Connex-S with a number of lanes between 32 and 1,024, clocked at 100 MHz w.r.t. the dual-core ARM Cortex A9 at 667 MHz with a total of two 128-bit Neon SIMD units.

GCC 8.3.0 offers little support for them. Even more, ARM's speculative hardware prefetcher is poor, not suitable for tiling.

The *MatMulBT-128/170/256.i16/i32/f16* benchmarks are actually implemented by the same C source function accepting two input and one output matrices of size $N \times N$, represented as 2D *variable-length arrays*, a feature available in C99 and newer standards of the C language. However, in case we have to perform tiling for matrices of size 512×512 or $1,024 \times 1,024$, we currently cannot use this parametric size kernel, mainly because the loop trip counts are not known at compile time so we cannot compute an optimal tiling statically. Also, the *DotProd* and *CtPop-Reduce* kernels take input data of parametric size, but we provide pointers as input instead of 1D *variable-length arrays* and, also, we do not need to tile the data because it is fitting the SPM.

For the *i16* benchmarks, we observe that we cannot accelerate the *DotProd* kernel, because it has a small arithmetic intensity, and the memory transfers to Connex-S become predominant. However, the other *i16* kernels achieve good acceleration, also in part because GCC 8.3.0 does not vectorize for ARM Neon our *i16* tests, which happens because it deems unprofitable the vectorization sum-reductions and multiplications of vectors. Notice that the *MatMulBT-170* benchmark for types *i16* and *f16* has a smaller speedup than the similar *MatMulBT-128/256* benchmarks, because we pad with 86 zero elements each row of each input matrix to make them both of size 170×256 to avoid performing partial reductions. We also note the speedup of the SAD-128 kernel is greater than the one of SSD-128, because ARM's branch predictor often misses when computing the absolute value, since the input data are random.

Most of the *i32* benchmarks achieve a subunitary speedup because of the big complexity of the *i32* operations emulated on Connex-S and because GCC vectorizes programs with *i32* type for ARM. To be able to accelerate these *i32* benchmarks on Connex-S, we should make it wider: For example, *MatMulBT-512.i32* on a Connex-S with 512 lanes should achieve a speedup factor of 1.9, which is about three times better than for 128 lanes.

We must not also neglect that the architectural speedup, which is the improvement of the number of cycles executed, in our case is greater than the reported speedups in Figures 2 and 3 basically by a factor of 6.67, given the frequencies of the CPU and the Connex-S processor are 667 and 100 MHz.

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 8.3.0 cannot vectorize floating-point operations for ARM Neon.

For these benchmarks, writing consecutively two blocks of 128 KB from the system RAM to the LS memory takes 1.265 ms. Also, to assemble an OPINCAA kernel takes an amount of time in the range 0.01–134.6 ms, the kernel *MatMulBT-256.f16* achieving the maximum with 308 Kinstructions. To amortize this overhead we cache the binary instruction stream we dispatch to Connex-S, for the same input data sizes, when running the kernel many times—100 or 1,000 times in our experiments. We also can precompute the binary stream of a kernel before the actual run, but such procedure is complicated and normally does not bring a serious advantage over caching when we run the kernel many times.

The generated vector assembler code is optimal for most benchmarks, except for *MatMulBT-128.i16*, *SSD-128.i16*, and *SAD-128.i16*, for which we can achieve 1.072, 1.072, and 1.049, respectively, times more speedup. For this, we need to avoid performing an unnecessary vector **add** strip-mining accumulation (see Listing 3) for dot-product calculation, since the trip count of the vectorized loop, 128, is equal in our experiments with CVL, the vector length of Connex-S. Similarly, for *MatMulBT-128.f16*, *SSD-128.f16*, and *SAD-128.f16* we take out the vector **add.f16**, which is time-consuming, because the assembler vector code has 500–700 instructions, and we achieve 1.66x, 1.4x, and 1.6x, respectively further speedup. We plan to add this optimization in the compiler removing the vector **add** accumulation in the future. *MatMulBT-170/256* cannot benefit from this optimization since the trip count of the innermost loop is different than CVL. The benchmarks *MatMulBT-128.i32*, *SSD-128.i32*, and *SAD-128.i32* cannot be optimized because the accumulation is performed actually on vectors of 64 *i32* elements so each dot-product performs two accumulations.

We can run the same generated C++ OPINCAA program on systems with Connex-S accelerators of different vector lengths since the number of lanes is an OPINCAA program environment variable, CVL, and we perform JIT vector assembling. In Figure 3, we present the speedups achieved by the generated OPINCAA programs when running on Connex-S processors of different vector lengths. In fact, since the CPU execution times does not vary with CVL, these graphs show the variation of the execution time on Connex-S when CVL varies. Note the experiments with 256, 512, and 1,024 lanes are performed with the Connex-S OPINCAA simulator since the Xilinx Zynq-7020 FPGA cannot accommodate these large Connex-S processor designs. We see how the performance of the benchmarks varies when increasing the vector length of the accelerator. This trend is not linear but asymptotic, especially visible for *MatMulBT-1024*, due to the big communication I/O overhead and the impossibility to speedup when increasing CVL the prologue and the epilogue of the vectorized loop. Note that for *MatMulBT-128/256/512* the speedup is stagnating after 128, 256, and 512 lanes, respectively, since the trip count of the respective vectorized loop becomes smaller than the vector length, and the rest of the lanes remain unused.

5. **Optimized Code Generation of Computer Vision Transformations using Sparse Matrices**

Modern computer vision image transformations can benefit from sparse input data in complex pipelines. However, such routines are difficult to implement efficiently with sparse matrices.

Therefore, we propose a compiler for a *Domain-Specific Language (DSL)* that computer vision specialists can use. The tool reads a mathematical function describing a transformation, relating the input and output image pixels, together with a generic C code template, and generates from them an efficient C implementation using sparse matrices. The compiler perform a novel code optimization to increase performance such that we traverse inorder the input sparse image. For this our DSL compiler needs to perform symbolic mathematical optimization on the equations of the function specifying the transformation inorder to invert the function.

We use our DSL compiler to help us develop the sparse version of a computer vision pipeline performing image alignment. We achieve an average sequential performance increase of 8.32 times on an x86 CPU at 2.66 GHz when using sparse matrices in various useful computer vision routines than when using dense images.

Also, we can accelerate some of these computations on our Connex-S wide vector processor achieving a minor performance increase w.r.t. just running on a CPU. However, we look to improve

performance by using a simple block sparse matrix representation starting from the observation that the sparse data in the frame is represented, by disks with radius of 2.5 (or 3) pixels.

The end goal with this chapter besides documenting an interesting and simple mathematical way to execute efficiently sparse matrix *stencil* or *map* McCool et al. 2012 CV computations starting from DSL specifications is to also show that such computations can be well accelerated on the Connex-S processor. We include these computations as part of our Connex-S runtime library, leaving their compilation from C to Connex-S as future work.

References

Bîră, Călin, Lucian Petrică, and Radu Hobincu (2013). "OPINCAA: A Lightweight and Flexible Programming Environment For Parallel SIMD Accelerators". In: *Romanian Journal of Information Science and Technology* 16.4.

Bocchino Jr., Robert L. and Vikram S. Adve (2006). "Vector LLVA: A Virtual Vector Instruction Set for Media Processing". In: *Proceedings of the 2nd International Conference on Virtual Execution Environments*. VEE '06. Ottawa, Ontario, Canada: ACM, pp. 46–56. ISBN: 1-59593-332-8. DOI: 10.1145/1134760.1134769. URL: http://doi.acm.org/10.1145/1134760.1134769.

Chandrakasan, A. P., S. Sheng, and R. W. Brodersen (1992). "Low-power CMOS Digital Design". In: *IEEE Journal of Solid-State Circuits* 27.4, pp. 473–484. ISSN: 0018-9200. DOI: 10.1109/4.126534.

DCAE (2019). *The Connex-S OPINCAA library, source code available at http://gitlab.dcae.pub.ro/research/opincaa*.

Kennedy, Ken and John R. Allen (2002). *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 1-55860-286-0.

Kodukula, Induprakas, Nawaaz Ahmed, and Keshav Pingali (1997). "Data-centric Multi-level Blocking". In: *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*. PLDI '97. Las Vegas, Nevada, USA: ACM, pp. 346–357. ISBN: 0-89791-907-6. DOI: 10.1145/258915.258946. URL: http://doi.acm.org/10.1145/258915.258946.

Kozyrakis, Christoforos (2002). "Scalable Vector Media-Processors for Embedded Systems". AAI3063439. PhD thesis. University of California, Berkeley. ISBN: 0493823174.

Kozyrakis, Christoforos E. and David A. Patterson (2003). "Scalable Vector Processors for Embedded Systems". In: *IEEE Micro* 23.6, pp. 36–45. ISSN: 0272-1732. DOI: 10.1109/MM.2003.1261385. URL: https://doi.org/10.1109/MM.2003.1261385.

Lattner, Chris and Vikram Adve (2004). "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, pp. 75–. ISBN: 0-7695-2102-9. URL: http://dl.acm.org/citation.cfm?id=977395.977673.

Lin, Haibo, Tao Liu, Huoding Li, Tong Chen, Lakshminarayanan Renganarayana, John Kevin O'Brien, and Ling Shao (2010). "DMATiler: Revisiting Loop Tiling for Direct Memory Access". In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT '10. Vienna, Austria: ACM, pp. 559–560. ISBN: 978-1-4503-0178-7. DOI: 10.1145/1854273.1854351. URL: http://doi.acm.org/10.1145/1854273.1854351.

McCanne, Steven and Van Jacobson (1993). "The BSD Packet Filter: A New Architecture for User-level Packet Capture". In: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX'93. San Diego, California: USENIX Association, pp. 2–2. URL: http://dl.acm.org/citation.cfm?id=1267303.1267305.

McCool, Michael, James Reinders, and Arch Robison (2012). *Structured Parallel Programming: Patterns for Efficient Computation*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 9780123914439, 9780124159938.

Mendonça, Gleison, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira (2017). "DawnCC: Automatic Annotation for Data Parallelism and Offloading". In: *ACM Trans. Archit. Code Optim.* 14.2, 13:1–13:25. ISSN: 1544-3566. DOI: 10.1145/3084540. URL: http://doi.acm.org/10.1145/3084540.

Naishlos, Dorit (2004). *Autovectorization in GCC. Proceedings of the 2004 GCC Developers Summit*. URL: http://people.redhat.com/lockhart/.gcc2004/MasterGCC-2side.pdf.

Nazaré, Henrique, Izabela Maffra, Willer Santos, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira (2014). "Validation of Memory Accesses Through Symbolic Analyses". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '14. Portland, Oregon, USA: ACM, pp. 791–809. ISBN: 978-1-4503-2585-1. DOI: 10.1145/2660193.2660205. URL: http://doi.acm.org/10.1145/2660193.2660205.

Nuzman, Dorit and Richard Henderson (2006). "Multi-platform Auto-vectorization". In: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO '06. Washington, DC, USA: IEEE Computer Society, pp. 281–294. ISBN: 0-7695-2499-0.

Patterson, David A. and John L. Hennessy (2017). *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Pouchet, L. N. (2014). *PolyBench: The Polyhedral Benchmark Suite*.

Ştefan, Gheorghe M. (2015). *The Connex-S Instruction Set Architecture*.

Şuşu, Alexandru E. (2020). "A Vector-Length Agnostic Compiler for the Connex-S Accelerator with Scratchpad Memory". In: *ACM Trans. Embed. Comput. Syst.* 19.6. ISSN: 1539-9087. DOI: 10.1145/3406536. URL: https://doi.org/10.1145/3406536.

Şuşu, Alexandru Emilian (2019). "Compiling Efficiently with Arithmetic Emulation for the Custom-Width Connex Vector Processor". In: *Proceedings of the Workshop on Programming Models for SIMD/Vector Processing*. WPMVP'19. Washington, DC, USA: ACM, 1:1–1:8. ISBN: 978-1-4503-6291-7. DOI: 10.1145/3303117.3306166. URL: http://doi.acm.org/10.1145/3303117.3306166.

The Linux Kernel Archives (2014). *https://www.kernel.org/doc/Documentation/networking/ filter.txt*.

The Polyhedral Model, http://polyhedral.info (2020).

Waeijen, Luc, Dongrui She, Henk Corporaal, and Yifan He (2015). "A Low-Energy Wide SIMD Architecture with Explicit Datapath". In: *J. Signal Process. Syst.* 80.1, pp. 65–86. ISSN: 1939-8018. DOI: 10.1007/s11265-014-0950-8. URL: http://dx.doi.org/10.1007/s11265-014-0950-8.