1 Introduction

Field Programmable Gate Arrays (FPGAs) are hardware devices that contain circuits that can be configured to implement the desired functionality. While they have traditionally been used for prototyping, there has been an increased interest in using these devices for computation. Part of the reason for this is that the computer architecture community is coming up against severe power restrictions - while Moore’s Law continues to hold and the density of transistors keeps increasing, hardware designers are finding it increasingly difficult to deliver power to these transistors and to dissipate the heat that they produce\[5\]. Recent work has shown that FPGAs can provide high performance computation with good power efficiency\[4\], which might allow these devices to be used to help available computational capability increase in the face of limited power budgets. An indication that the mainstream processor community is beginning to embrace FPGAs is Intel’s recently released Stellarton\[9\] platform, which combines an Atom processor and a reconfigurable FPGA fabric (which is based upon an Altera product) within a processor package.

While the capability of FPGAs to effectively perform computation has been demonstrated, actually using these devices in such a fashion remains a challenge. Languages that are typically used to create FPGA applications such as Verilog and VHDL present a very different programming model from that which software developers generally use, and the learning curve for these languages is steep. In addition to overcoming the language hurdle, creating any significant application requires the ability to communicate between the chip and the outside world and use more memory than is available locally. Manufacturers of FPGAs do provide such communication interfaces, but not only do these interfaces need to be manually instantiated for each application, but the mechanisms used to interact with these interfaces varies between device manufacturers, and can even vary between different devices from the same manufacturer. After instantiating these interfaces, the application designer must also instantiate a data path that allows data transport to occur, and control logic to ensure that data is sequenced properly.

One answer to these issues has emerged in the form of CoRAM\[3\], a proposed design for a standard memory architecture for FPGAs. CoRAM provides a standard memory interface across device manufacturers, and an integrated network on chip for data delivery within the reconfigurable fabric. Sequencing is performed via control threads that the developer can writes using a C-like language, which can include overlapping asynchronous I/O operations and control flow. CoRAM can be implemented in hardware within future FPGAs, but can also be instantiated as a soft design in current generation devices. A paper (to be presented in the 2012 Internal Symposium on FPGAs) shows that CoRAM does allow the full capabilities of the FPGA to be exploited when implemented intrinsically within the device, and that the soft version is effective for some workloads, particularly for workloads that are not bandwidth or latency limited. A question remains as to whether or not CoRAM is amenable to full system compilation, and this work is intended as a step towards answering this question.

ISPC is the Intel SPMD Program Compiler\[10\]. Like OpenCL\[11\] and CUDA\[12\], it allows users to write loop based programs to indicate that loop nests should be parallelized in a Single Program Multiple Data (SPMD) system. The difference between ISPC and CUDA and OpenCL is that ISPC is intended to be compiled directly to Single Instruction Multiple Data (SIMD) processor instructions such as Intel’s SSE or AVX. The difference between SPMD and SIMD is that SPMD allows parallel instruction lanes to be individually enabled or disabled, so some control flow divergence can occur. ISPC is somewhat simpler than OpenCL or CUDA in that ISPC does not contain various levels of local memory like the other languages do, and does not contain a concept analogous to thread blocks. ISPC is distributed as open source LLVM based compiler, which made it attractive as a base technology for this work.

2 Related Work

High Level Synthesis is the general name for techniques to produce a hardware design (for use on an FPGA or an ASIC) from a high level description such as a software program\[7\]. A number of systems that attempt to provide High Level Synthesis capabilities exist, but all existing systems have some limitations.

AutoESL\[13\] is a commercial system that is distributed by Xilinx, which also produces FPGA devices. It con-
verts C code to a state machine that is either compiled to a finite state machine or run on an embedded microprocessor. This tool does require some manual restructuring of the code, particularly if off chip memory is to be used, and does not attempt to automatically generate parallel implementations of the provided software, although it does allow the user to annotate the code to allow it to be pipelined[14].

Altera’s C2H workflow[15] also enables compilation of some parts of a C program to Altera’s FPGA parts. It is not intended for whole program parallelism, and while it can schedule parts of the code that it compiles to run in parallel, enabling parallel computation is not the goal of the compiler.

LegUp[2] is a recently developed system for whole program compilation to FPGA. It is designed to be flexible and support a large subset of the C language, but does not produce highly pipelined or parallel programs, nor does it support floating point operations.

ROCCC[16] is an open source product that can produce highly pipelined compute kernels from C source, and does include support for floating point operations, but includes very limited support for control flow, and does not support accessing memory, although it can create address generation hardware.

BlueSpec[17] is a Haskell-derived language that compiles to Verilog code. This code is typically not a complete system, but consists of modules that are used within a larger design. While BlueSpec is easier to write than typical hardware design languages, it is a functional language that can be daunting to new users.

FCUDA[8] is system to allow programs written in CUDA to be compiled to an FPGA. While it does produce a functional system, it does require that application developers annotate their code to indicate exactly how it should be implemented on the FPGA.

OpenMP is a framework for parallel programming that allows loops to be annotated to indicate how parallelism occurs. A project from the Barcelona Supercomputer Center[1] did seek to target FPGAs from OpenMP programs, but like FCUDA required the programmer to be extremely explicit in how the kernels are described and how data is transferred. A project from the National University of Singapore[6] had similar goals, but did not appear do anything to handle data marshaling between the FPGA and off chip memory - it appeared to focus solely on programs that generated data.

It appears that Altera is working on mechanisms to compile OpenCL code to the FPGA[18], but no official information has been published at this time.

3 Flow and Architecture

The general approach taken during the implementation of this work was to leverage the CoRAM framework and an existing research project that seeks to compile a subset of programs that can be converted to LLVM byte code to FPGA systems. The hope was that the constructs provided by the ISPC language would make it easier to produce parallel systems, since ISPC program are somewhat more limited than C programs, and parallelism is explicitly annotated by the application developer. The intent is to target the current soft implementation of CoRAM, which can implement complete systems on current generation FPGA devices. An overview of the compilation workflow is illustrated in figure 1.

The steps performed during compilation are:

1. Compile the ISPC module
2. Post-process the ISPC compiler output
3. Partition the program into control threads and kernel operations
4. Generate hardware kernels
5. Generate a top level module
6. Generate the hardware system

4 Implementation

A detailed description of each step and how it is implemented follows.

4.1 ISPC Program Compilation

The first step in the process is to compile the ISPC module. Full ISPC programs consist of two parts: an
ISPC module and a C module. The ISPC module contains all of the ISPC code to be executed, which are exported to the rest of the system by generating an object file and a C style header file. The object file follows the standard calling conventions of the operating system being used, allowing the the code to be linked like any other object file using a standard linker.

As ISPC was intended to produce SIMD code, an intrinsic part of the code generator within it handles tracking lane masks to allow execution of SPMD programs. ISPC natively supports Intel’s SSE2, SSE4, and AVX extensions to the x86 instruction set. The original plan was to use the stock ISPC compiler, possibly hooking into its parsing mechanisms to determine how the program was annotated for parallelism, and use its LLVM byte code output with the CoRAMify partitioning pass and CoRAM Compiler system generation pass. This tasks turned out to be more difficult than anticipated. The first difficulty was that the current release of ISPC is designed for LLVM 2.9, while the CoRAM passes use LLVM 2.8. LLVM 2.9 is the current version of the LLVM compiler, although version 3.0 is scheduled to be released shortly. Version 2.9 LLVM byte code cannot be read by version 2.8 passes due to format differences in both the binary and assembly file formats. An attempt was made to port the two CoRAM passes to version 2.9, but difficulties in this conversion made it more feasible to back port ISPC to version 2.8. The ISPC documentation did mention that LLVM 2.8 had difficulties with the vector instructions that it produced, but since the ultimate intent was not to use vector instructions these difficulties were deemed not important.

The next issue encountered was with the output targets supported by ISPC. The intent was to have ISPC produce scalar files and retain the annotations that indicated parallelism. This was required because the hardware kernels are generated using the ROCC[16] compiler, which requires using LLVM’s C back end to produce code to use with it. The C back end does not support vector operations. Unfortunately, ISPC only supports vector output. One difficulty in making ISPC support scalar output is that the idea of tracking and using lane enable masks is pervasive to the ISPC code generator, and it appeared that removing the lane mask would require largely rewriting the several thousand lines of ISPC’s code generator. Since code generation can be complicated, and ISPC code generation was based upon an abstract syntax tree that was not thoroughly documented, a new ISPC target was created to enable generating scalar code. As previously mentioned, it was infeasible to directly generate scalar code, so the approach taken was to generate 1-wide vector code instead, and convert this code to scalar code downstream in the workflow.

Creating the 1-wide vector target was relatively straightforward, although it was necessary to implement ISPC’s standard library for 1-wide vectors. This library consists of LLVM assembly code that implements load, store, permutation, and mathematical operations. The mathematical operations were required because the goal was to support programs that include floating point computations, and ISPC programs cannot link to the standard math library. LLVM does support floating point intrinsics, so it was possible to implement the floating point operations by calling these intrinsics.

One other minor complication is that by default, ISPC generates both an exportable version of the main function to be called and an internal version, which takes a lane mask as a parameter, and contains mangled function and variable names. ISPC was modified to suppress generation of the internal version of the exported function.

4.2 ISPC Post Processing

The second step in the process is post processing the byte code produced by ISPC. Standard LLVM passes are used to convert the byte code to Single Static Assignment (SSA) form, to remove dead code, to move loop invariant code out of the loop that contains it, and to perform other optimizations. It was hoped that the standard LLVM passes would be able to completely remove the lane masks which do not do anything on 1-wide vectors, but this turned out not to be the case. In most cases, this is not an issue, but it did turn out to be a problem for one attempted work flow.

After using LLVM standard passes, two custom passes are used prior to the CoRAMify partitioning pass. The first pass is a devectorization pass. The previous section described how it was necessary to generate 1-wide vector code rather than scalar code. The devectorization pass converts the 1-wide vector code into scalar code. It traces through every instruction, tracking which instructions are vector instructions. Vector loads are replaced by scalar loads of the same data, and every subsequent instruction that uses vectors is replaced by an identical scalar instruction using the corresponding scalar operands. LLVM provides user instructions, which makes it easy to trace through instructions as they are devectorized up until data is stored back to memory. The one complicated instructions are Phi nodes, since they in-
troduce circular use patterns, and these are resolved by creating fake instructions when Phi nodes are first encountered and cleaning them up after all other instructions have been processed.

The second custom post processing pass is a de-casting pass. Since ISPC uses a standard library to process various instruction types, there are many instances in which pointers are cast from their native types to generic void pointers to integer pointers, and then the data is cast back to the original type. These casts don’t incur a run-time overhead as they don’t actually do anything, but can potentially confuse later analysis passes. The de-casting pass traces through all cast instructions, and removes any cast-recast operations that it finds.

4.3 CoRAMify Pass

The CoRAMify pass is responsible for analyzing the program and partitioning it into kernel operations, which as previously mentioned are converted to hardware using a package called ROCCC, and all other operations, which are converted into multiple CoRAM control threads. The pass operates as follows:

1. Instructions are traced to determine if they are data instructions, control flow instructions, or memory instructions. Memory instructions are used as addresses, control flow instructions are branch instructions or instructions that influence branch instructions (such as comparison instructions), and data instructions are instructions whose results are ultimately stored to memory.

2. Memory instructions are relocated - load instructions are moved as early as possible and store instructions are moved as late as possible. Instructions that calculate memory addresses are grouped with the memory instructions that use them.

3. The data instructions are moved into a new function, which is later converted to C for use with ROCCC.

4. Constants such as memory addresses and loop bounds, which are passed to the CoRAMify pass, replace parameters to the function that is being converted and global values used by the function.

5. Each input or output to the new function is converted into its own stream. The original function is then broken up into multiple functions, one for each stream. This makes it easier to reorder loops and perform other operations on the code without worrying about unintended side effects.

6. The loop nests for each function are analyzed to determine how the memory accesses in each stream at that loop nest. At this point, accumulations, where a variable is both read and written, are inferred. LLVM’s Scalar Evolution Analysis framework is used to perform this analysis, although much work went into supplementing the information that it provides.

7. If there is an accumulation, and that accumulation happens in an inner loop nest which would prevent the application from exploiting parallelism, then the loops are re-ordered, and stream accesses rewritten, for all functions. It is necessary to rewrite all of the functions at this time to ensure that the correct data is delivered to the hardware kernels.

8. Each stream is processed to determine what memory access patterns it exhibits in order to group memory accesses into larger blocks and reduce memory bandwidth requirements. In particular, the CoRAMify pass looks for sequential accesses, repeated accesses, and places where arrays may need to be transposed. Sequential accesses are combined into a single, larger access.

9. The system detects how many compute kernels to instantiate. The number of kernels that is instantiated depends on the amount of parallelism found and a maximum number of kernels that is specified as a parameter to the system.

10. The system instantiates CoRAM streams, and converts all memory reads and writes in the control threads to stream operations.

11. The descriptor file containing information about the CoRAM streams is created, which is used to generate the top level module.

4.3.1 Partitioning Example

As a concrete illustration of how the partitioning pass works, Listing 1 contains the complete source code for an ISPC implementation of a blocked matrix matrix multiply, and is in fact the exact ISPC source code used by the matrix-matrix multiply test case. The function is decorated with the "export" keyword, indicating that this function is meant to be called by the main program,
export void mmm_ispc ( uniform float A[] , uniform float B[] , uniform float C[] , uniform int size , uniform int blocksize ) {
    for ( uniform int i0 =0; i0 < size ; i0 += blocksize ) {
        for ( uniform int j0 =0; j0 < size ; j0 += blocksize ) {
            for ( uniform int k0 =0; k0 < size ; k0 += blocksize ) {
                for ( uniform int i1 =0; i1 < blocksize ; i1++ ) {
                    uniform int i = i0 + i1 ;
                    for ( uniform int jc =0; jc < blocksize ;
                        jc += programCount ) {
                        int j = j0 + jc + programIndex ;
                        for ( uniform int k1 =0; k1 <
                            blocksize ; k1++ ) {
                            uniform int k = k0 + k1 ;
                            C[ i * size + j ] += A[ i * size + k ] * B[ k *
                                size + j ] ;
                        }
                    }
                }
            }
        }
    }
}

Listing 1: ISPC source code for a blocked matrix matrix multiply

which will cause ISPC to include its declaration in the header file that it creates. The function takes parameters for the a and b input arrays, and the c output array. The output array is assumed to be zero-initialized in this implementation. Parameters also indicate the size of the arrays and the desired block size - so a different block size can be implemented by simply changing this parameter.

This implementation is almost exactly the same as a canonical implementation of a blocked matrix-matrix multiply in C. There are a few minor differences:

1. The parameters to the function, and all loop iteration variables except for the loop on line 7, are decorated with the "uniform" keyword. This indicates that these variables take the same value in all execution lanes.

2. The loop starting on line 7 is incremented by "programCount" rather than 1, and the actual variable used for computation is the sum of that variable and "programIndex". These indicate that these variables will be used by different execution lanes, and in particular each execution lane has a unique "programIndex" variable, which is a number between 0 and the number of lanes - 1.

3. This particular implementation assumes that both the matrices themselves and the blocks are square, although nothing in the compiler requires this.

The analysis of the code occurs as follows:

1. The code in line 11 is identified as the kernel code. Line 11 actually consists of a number of operations - four address calculations, three loads, one store, and one floating point multiply and one floating point add.

2. The loads and associated address operations are moved to the top of the basic block containing the loop body, and the store and its address operations are moved to the bottom of this block.

3. The floating point multiply and add are moved into a separate function. This function is actually implemented in such a way that addresses for the input and output variables are passed to it.

4. The main function mmm_ispc is replaced by a version that takes no parameters, and expects A, B, C, size, and block size to be global variables. These global variables are then placed with the values provided to the compiler. It is important to note that the user is required to provide values for these variables at compile time in order for the compiler to work correctly.

5. The main function is replaced by four versions - one that reads A, one that reads B, one that reads C, and one that writes C. The fact that c is both read and written to is stored at this point.

6. The analysis of each variable is as follows:

   - a is read sequentially in the loop on line 9. A is invariant in the loop on line 7 (since j does not appear in the expression for A), and is read in a non sequential form in the loop on line 5. Similarly, it is sequential in the loop on line 4, invariant in the loop on line 3, and non sequential in the loop on line 2.

   - B is read non sequentially in the loop on line 9, sequentially in the loop on line 7, and invariant in the loop on line 5. Similarly, it is non sequential in the loop on line 4, sequential in the loop on line 3, and invariant in the loop on line 2.

   - The memory access pattern for C is the same for both the read and write accesses, and the
compiler has determined this fact by this point. It is invariant in the loop on line 9, sequential in the loop on line 7, and non sequential in the loop on line 5. It is invariant in the loop on line 4, sequential in the loop on line 3, and invariant in the loop on line 2.

7. During this step, the compiler has determined that the accumulation over c occurs at the innermost loop nest, which would prevent parallel operation. The loop is moved outwards two steps, right before the accumulation at line 4. It stops at this point because it has determined that a sufficient amount of parallelism is now available. The progression of all variables are re-written to match the new loop order.

8. At this point the compiler attempts to roll up variable accesses as much as possible.

- A is now invariant at the loop on line 9, non sequential at the loop on line 7, and sequential in the loop on line 5. The loops on line 9 and 5 are modified so that their iteration count is 1 (since the invariance is handled by repeating the number of items seen up to that point - 1 in this case - and the sequential accesses are converted into one access with a for a larger number of items). Since there is a non-sequential access inside the sequential access, the need to transpose a is noted.

- B is now sequential in the loop at line 9, invariant at the loop at line 7, and non sequential in the loop at line 5. The innermost loop is converted to a access. The invariant loop is handled by noting that one long access of items needs to be repeated. The outer loop is left alone because it is non sequential.

- C is now sequential in the loop at line 9, non sequential in the loop at line 7, and invariant at the loop at line 5. The sequential access is converted into a larger access. The invariant access at line 5 is converted to a single loop iteration, and since the next outer access - at line 4 - is also invariant, it is also rolled into the accumulation. The accumulation will accumulate everything inside it, or all items transferred in the loops at lines 7 and 9.

One thing to note is that if the block size is equal to the matrix size - if there is only 1 block - then the compiler can determine that the non sequential accesses are actually sequential accesses of larger blocks (once the sequential accesses have been modified) - in this case there is only one access for each variable after transformation, and the entire matrix is sent in a single transfer.

9. At this point the compiler sees that there are as many operations as items in the block that can be executed at once. It will instantiate up to this many kernels, and also up to the maximum number of kernels passed to the compiler. The optimal number of kernels is actually the number of items in the block / 20, since the compute kernel has a latency of 20 cycles. In practice, the experiments used a number of kernels equal to the number of items in a block row.

10. Stream instantiation is a simple transformation that converts the loads into a transfer of the number of items inferred in step 6.

11. The descriptor file contains the id of each stream, the type of stream it is (regular stream, repeat, transpose, or accumulation), the number of kernels, and the desired external data path width. An example of the system descriptor file is in Listing 2. This file was generated 128x128 matrices, 64x64 blocks, a maximum of 64 compute kernels, and a maximum external data path width of 64 bytes. The system that is instantiated is displayed in Figure 2. This diagram shows only the components generated by the CoRAMify compiler, the Kernel Generation Step, and the Top Level Module gen-

| NumKernels:64 |
| ForceSize:64 |
| Thread:mmm_ispc |
| TransposeWriteFifo:0,64,1,64,64,262144 |
| Thread:mmm_ispc_0 |
| LoopWriteFifo:1,63,1 |
| Thread:mmm_ispc_wr |
| ReadFifo:3,64 |
| Thread:mmm_ispc_1 |
| WriteFifo:2,64 |
| AccumulatorPair:2,3,8128,64 |

Listing 2: System Descriptor file for Matrix Matrix Multiply with 64x64 blocks
4.4 Kernel Generation

The kernels are generated by using LLVM's C backend to generate C from the byte code representing the kernel computation and passing it through the ROCCC kernel generator after a minimal amount of text processing to clean it up. ROCCC generates fully pipelined VHDL compute kernels that support floating point operations that use FPGA vendor provided compute cores. ROCCC is configured with the latency information for each floating point operation, and generates registers to ensure that the number of cycles needed for the operations is maintained. ROCCC handles control flow within the cores via predication, and creates modules with inputs and outputs specified by the data streams created by the CoRAMify pass. The compute kernels have no information about data access patterns and are stateless except for data moving through their pipelines, initiating computation whenever all of their inputs are ready and able to produce 1 calculate per clock cycle.

4.5 Top Level Module Generation

The top level module is generated from a descriptor file that is generated by the CoRAMify pass containing information about the various queues in the system along with the source code of for the compute kernel, which is used to determine the names and sizes of each input and output. The top level module generation script creates a BlueSpec module that instantiates the CoRAM fifos used by the input and output queues, the specialized buffer structures and state machines that implement each of the queues according to the detected access pattern, the appropriate number of instances of the compute kernel as determined by the CoRAMify pass, and the logic necessary to steer the data between the queues and the compute kernels. This top level module is fairly simple - it initiates computation whenever the input structures it contains indicate that all inputs are ready, and retires computations whenever computations are complete. The module generation script makes sure that there is enough buffer space available for all kernel outputs.

After the module generation script is complete, the top level module is run through the BlueSpec compiler, along with any library modules that it uses (such as queue handling buffer modules), to produce verilog code for the system generation step.

4.6 System Generation

The system generation step takes the verilog modules generated by the top level module generation step and the LLVM byte code for the control threads generated by the CoRAMify step, and builds a complete system using the soft CoRAM platform. This step is implemented as a LLVM pass that can parse through the enough of the verilog modules to infer modules, communication ports, embedded memories (BRAMs for the Xilinx platform), and the connections between them. This LLVM pass produces a state machine implemented in Verilog for each control thread. It also instantiates the DRAM interface, the network on chip (with enough input ports from DRAM to use all of the available memory bandwidth), and an instance of the top level module instantiated in the previous step. All of these modules, along with the VHDL modules produced by the Kernel Generation pass and the hardware cores that it uses for floating point computation, comprise a complete system.
that can be compiled into a bit file for an FPGA or simulated using an HDL simulator.

5 Experiments

The experiments below in simulation modeling an FPGA system similar to that provided by Xilinx’s ML605 prototype board. This board is readily available and contains a relatively recent generation FPGA chip (A Virtex-6, which was the current generation of Xilinx’s products until very recently - the Virtex-7 line of chips has been released, but are not yet readily available). It also contains a single DDR-2 memory interface. The simulator used was Xilinx’s iSim version 13.3, which is the latest version available of this cycle accurate HDL simulator.

5.1 Matrix-Matrix Multiply

The first experiments performed were with a matrix-matrix multiply, using the ISPC code documented in Listing 1. The goal of this experiment was to see if a fully automated compilation workflow could saturate the computational ability of the FPGA. This is a follow up on an earlier experiment that determined that a hand tuned version of a matrix-matrix multiply could saturate an FPGA using CoRAM (with 1024x1024 matrices and 128 processing elements, although a very different parallelization pattern), demonstrating the feasibility of the platform itself. The tests varied the number of compute kernels between 64 and 128 and the size of the matrices between 128x128 and 512x512, and set the block size equal to the square of the number of compute kernels (for example, for 64 compute kernels, 64x64 blocks were tested).

Figure 3 shows the raw results achieved by the matrix-matrix multiply. The number of items along each dimension of the matrix is doubling with each step in the graph. Because the algorithm used here is the simple \( n^3 \) algorithm for multiplying two matrices, the total number of computations to be performed increases by a factor of 8 as the matrices get larger. However, the number of cycles taken as the size of the matrices get larger increases by a much smaller factor. This indicates that the overhead of the data transfers is being amortized by the block structure - the application is able to successfully overlap computation with data transfers, and keep the highly pipelined compute kernels busy. Figure 3 shows the throughput that was achieved, assuming a 200 MHz system clock, which is a reasonable speed for the board being tested. Each kernel can reach a theoretical maximum of 2 flops/cycle, since it performs a floating point multiply and a floating point add. Thus the theoretical maximum throughput for 64 kernels at 200 MHz is 25.6 GFlops/s, and maximum possible throughput for 128 kernels is 51.2 GFlops/s. As the size of the matrices increases, it is evident that the achieved throughput is approaching the theoretical maximum.

Figure 5 shows the efficiency achieved by the computation. This is defined as $\frac{\text{Achieved Throughput}}{\text{Theoretical Throughput}}$ = $\frac{\text{Minimum cycles for computation}}{\text{Actual Cycles for computation}}$. The graph shows that the application approaches 100% of the peak possible throughput as the matrix size increases, and in fact shows that 95% of peak efficiency can be reached when there are at least 4x4 blocks (64x64 blocks with 128x128 matrices or 128x128 blocks with 512x512 matrices). Above this point, the efficiency continues to increase with the increase in matrix size, but much more slowly.

5.2 Vector Addition with Indirection

The second experiment tried was a simple vector-vector addition with an indirection. The source code for this application is included in Listing 3. The output vector is the sum of two input vectors, with the a level of indirection used to index one of the input vectors. The goal of this experiment was to demonstrated that the compiler could support indirection, and to determine how it performed.

This experiment was tried with 8 compute kernels, with vector sizes varying between 64 and 512 items. Figure 3 shows the number of cycles taken by the application as it ran. In this case, the work to be performed...
increased linearly as the vectors increased in size, as did the execution time. This indicates that the application was not able effectively overlap computation with memory accesses. The reason for this is that the latency caused by the indirection dominated the actual computation.
6 Discussion and Future Work

ISPC is an interesting language to use, and looks like it will be an effective mechanism to create SIMD code in the future. The availability of an open source LLVM based compiler made it an attractive option for this work, compared to the CUDA or OpenCL languages, for which no open source fully operational compiler was available.

One piece of information that is provided by the ISPC language that did not turn out to explicitly be useful is the "uniform" keyword. The language uses this keyword to indicate that data does not change in the loop that implements the parallelism, which means that it is invariant at that level. The scalar evolution framework within LLVM should be able to infer this information automatically, so this keyword is not strictly necessary. However, this information does make the compilation process easier, especially in the case of a larger program with many function calls, where tracing them could be tricky. Such a system could have functions where a function is called in such a fashion that a parameter is sometimes varying and sometimes uniform, although the compiler should be able to tell this as well.

Given the uniform keyword, there are two places where the compiler can use this information. One place is a traditional use of knowledge about invariant information - the instructions that compute the invariant data can be moved out of the loop, reducing the number of instructions in it. LLVM contains standard compiler passes for performing this transformation that does not require the uniform keyword. The other use of this data is specific to SPMD programs. SPMD programs must maintain masks to indicate which lanes are currently active and which are not, and they must contain special mechanisms to handle divergent control flow. If the variables used to determine control flow are uniform, then the compiler does not have to instantiate the logic for handling divergent control flow, so can produce much better code. This optimization is not relevant to an FPGA implementation because an FPGA implementation can actually handle divergent control flow within the hardware kernels because they are actually independent of each other - the kernels actually implement control flow via predication, which actually also means that the number of cycles taken by the kernel (at least in ROCCC’s implementation) does not vary depending on which control flow path is taken.

There are a number of places where it would make sense to extend this work in the future, most of which...
are enhancements that would allow it to support a larger variety of applications.

It is unlikely that much could be done to reduce the latency for the indirection for the vector addition application. The exact same issue cropped up with the manually created implementation of a sparse vector/matrix multiply application for CoRAM, and the conclusion of that work was that the indirect workload would need a hardened version of CoRAM, which would have lower memory access latency, to work well. However, CoRAMify pass does not perform any memory stream detection for indirect memory accesses. Adding support for these accesses might help a bit.

It would make more sense to focus effort on the image filtering application, which has enough data reuse that it should work well. The two issues encountered in implementing it - the remnants of the lane mask in ISPC's output and the offset memory access - are not insurmountable. It should also be possible to detect and implement a memory buffer to handle a sliding window access, which would greatly reduce the memory bandwidth requirements for such an application. Even without supporting that pattern, there should be a way to detect that this pattern has multiple sequential streams that are accessed at the same time and handle them pretty well. The Matrix-Matrix Multiply application could be modified to prevent the output buffer from needing to be zero initialized and read at the start of the computation by initializing the input buffer to zero directly. However, some implementations of Matrix-Matrix Multiply use multiple levels of blocking, which would require the return of reading in the input buffer. Additionally, this work has shown that high efficiency can be reached when there are at least 4x4 blocks in the matrix. In this situation, computing a block of C involves bringing in the block, writing out the block, and reading 4 blocks of A and 4 blocks of B. An implementation that was more efficient would not bring in the block of C, and would therefore transfer 9 blocks instead of 10 per output block. This indicates an overhead of 11% for 4x4 blocks (5.8% for 8x8 blocks), which, while high, is not important given that the computation is not bandwidth limited.

7 Conclusions

This work introduced an FPGA compilation workflow for the ISPC programming language. It demonstrated that ISPC is a good language for writing software that will be compiled to FPGA, at least for some workloads. While constructs such as the uniform keyword were not used directly, they allowed the compiler to produce code that explicitly has parallelism within it, and some features of ISPC - such as the requirement that function parameters not alias each other - make explicit ideas that had previously been implied.

Three applications were tested, one that was extremely successful, one that worked correctly but not efficiently, and one that did not work. The issues that prevented the image filtering application from working were not insurmountable, but time constraints prevented them from being resolved. It is likely that this application will be completed in the future, either with the ISPC language or for the CoRAMify compiler pass, since it is one that should work well on the FPGA.

References


