Spring 2020 End of Semester Report: Accelerating a Coherent Ising Machine using FPGAs

Aryaa Vivek Pai
McMahon Lab,
School of Applied and Engineering Physics
Cornell University
(Dated: May 14, 2020)

The Coherent Ising Machine simulator developed by Mahon Lab aims to solve discrete optimization problems known as Ising problems [1]. This project focused on accelerating the simulation of the Coherent Ising Machine using a FPGA is a continuation of the project by Darren Schachter in Fall 2019. The advantages of using an FPGA to accelerate the simulation is its flexibility and capacity to create customized algorithm specific designs. During the previous semester, a stripped-down version of the simulation program was created in OpenCL. Due to lack of documentation, a decision was taken to switch to C++ programming. This paper outlines the progress made on during Spring 2020 semester in accelerating the CIM simulator. It explains the various decisions taken to optimize the program including trade off between sequential and parallel architecture as well as computation and storage.

I. THE OPENCL MODEL

The OpenCL computation model is a parallel programming model used in this project to accelerate the CIM simulator code on the FPGA. OpenCL programs are divided into host code and kernel code [2]. The OpenCL platform model specifies that there is one host such as a CPU which coordinates the execution of multiple compute devices such as FPGAs which execute the kernels. The OpenCL execution model defines the environment configuration used by the host and the concurrency model used to configure kernels. According to the concurrency model, a two-dimensional dataset such as an N by N matrix, is broken in NxN work-items. A certain number of work-items that computed together are grouped into workgroups. Thus, taking complete advantage of the parallel architecture. The Kernel programming model how the concurrency is mapped to the hardware, during the build process [3].The Memory model specifies the use of private memory and shared memory between the host and devices.

A. Host Code

The Host Code is the program that runs on the host computer and delegates tasks to various compute devices, the FPGA instance in our case. The host code is written is C++ using the OpenCL API. The main tasks of the host application include setting up the platform, allocating and transferring buffers to the device, launching execution of the kernel on FPGA, reading output buffers from the device memory and clean up post processing [4].

As a part of the setup, the host code primarily finds the available computational resources and reserves them to complete current workload. A platform consists of the single host that communicates with multiple kernels. Once the host connects to the first discovered platform, it will then connect to a compute device. In our case this is the FPGA instance known as `CL_DEVICE_TYPE_ACCELERATOR`. A context is a grouping of compute devices and memory objects assigned to execute kernels and data transfers to them. The host now defines a context and command queue that holds the list of specific tasks and memory transfers assigned to the device in the context. Next, the program to be executed is created from the source files and complied using the binary APIs. As per the OpenCL standard, kernels for FPGA are compiled beforehand, so in this case the host will use the binary files instead of the source files.

The host needs to allocate and transfer buffers to the FPGA device. In the OpenCL memory model, there are four different disjoint memory spaces for the kernel and a host memory. The __global and __constant memory are shared by both the kernel and the host. The global memory buffers that transfer data between the host and kernel are defined in __global while read only variables such as the matrix data size is defined in __constant memory. All arguments used in kernel functions are declared in __private memory. When arguments such as matrices and vectors which are kept track with pointers, are declared as pointers in __private space which point to an object in __global space [5]. The __local memory associated with the device contains variables used by all work groups [6]. The host first creates buffers in the host memory, accessible only to the host, and stores input data for operations in them. This input data can be read from another file or assigned with arbitrary values. For the purpose of testing the matrix vector product program, the matrix and vector were with values depending on the row and column of the element. In the actual CIM simulator, values will be read from a .txt file to avoid building the code for different matrices. This functionality has already

been tested in the code in CIM git hub. The host then creates the global memory buffers to transfer data with the kernel and specifies each buffer with size, context assigned to it as well as access permissions (R/W) for the kernel. The buffers used for kernel's input arguments have read only access while the buffers used to get the output from the kernel is provided write access. The host transfers data from the host buffers to the global buffers, which can be accessed by the devices. Until this data transfer is complete the host stalls other procedures.

In the next step, the kernel to be executed is

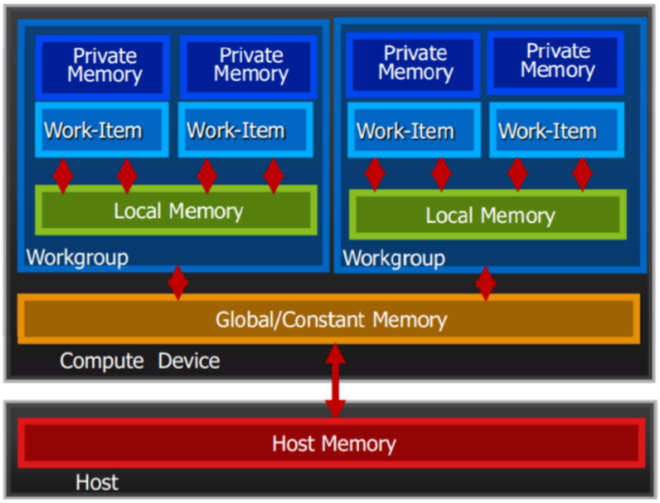

FIG. 1. The OpenCL memory model.

created and the parameters are set to the previously defined global buffers. As per the OpenCL execution model, in an NxN computational domain, each independent element called a "work-item". A "work-group" consists of multiple work items executed in parallel depending on the number of computational units available. The dataset is broken in the multiple work groups. The kernel is executed for each work group so that all the work items in the dataset are computed. To execute the kernel, the host schedules the task on the command queue for the FPGA device. Since a FPGA does not give the computational units capable of directly executing kernel code, the user defines the number of computational units used for the kernel execution. These design decisions that can optimize the computation time still need to explored. During execution, the kernel will write to the output buffer in the global memory space. Once the kernel has been executed, the host needs to explicitly initiate transfer from the global memory to the host memory. The task to read from the global buffer to the host memory is placed on the FPGA's command queue. Like before, during this data transfer all other processes are halted by the host to preserve memory consistency. The same process is repeated for executed of other kernels. The data received is transferred to a text file for testing and debugging. At the end of the host code, all the resources are released.

B. Kernel Code

Kernel code is the compute intensive part of the whole program which mean to be accelerated on the FPGA instance. Kernel is the program that runs on the compute device. Decisions regarding the micro-architecture of the FPGA and optimizations are specified in the Kernel using pragmas and attributes supported by Vivado HLS [7]. In the SDAccel environment, Kernel code can be written in C or C++, OpenCL or RTL. Last semester this project was carried out with an OpenCL kernel because of its standardized API designed to support parallel computing, complex device mangagement and high level of abstraction. However, it was difficult to optimize the code and use the features in OpenCL due to the lack of documentation for designing OpenCL programs for FPGA hardware. A decision was taken to switch to C++ kernel owing to the abundance of resources available as well as Xilinx tutorials. An advantage of C++ kernels is support for fixed point arithmetic. The Ising Simulate currently use floating point numbers for computation but using fixed point arithmetic will save the power efficiency and area significantly while keeping the same level of accuracy, thus help accelerate the simulator. The shift to C++ kernels has made it easier to find kernel examples and debug the code.

II. OPTIMIZATION OF THE MATRIX VECTOR MULTIPLICATION PROGRAM

To understand how to program kernels in C++, a matrix vector multiplication kernel was created. The host code was also rewritten to interface with this new kernel. The purpose behind creating this program was to understand writing kernels in C++ as so to eventually accelerate the Ising Simulator using C++ kernels applications run on the FPGA. Since the main operations involved in the Ising Simulate Architecture are multiplication and addition operations on matrix and vectors, it was appropriate to create matrix vector multiplication program. A similar approach was used last semester while writing kernel code in OpenCL. The basic matrix vector multiplication code consists of a nested loop. The Figure 2 shows the implementation of the matrix vector multiplication in sequential architecture resulting from an unoptimized code. The whole idea of accelerating the code on a FPGA take advantage of the highly distributed and parallel FPGA architecture to provide a performance boost compared to running on a CPU. Different versions of the code were created using techniques documented below for optimization. A System Estimate Report was generated for each version for different sizes of datasets. The scaling as size increased, timing, latency and re-

source utilization such as DSPs and RAM were compared for each of the runs [8].

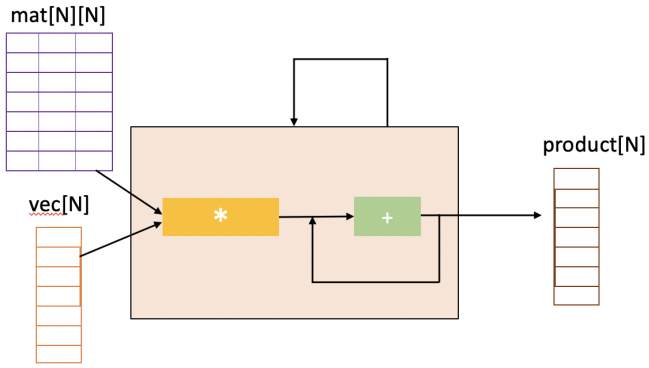

FIG. 2. The implementation of a basic matrix vector multiplication. Each iteration involves a multiplication and addition. The input matrix - mat is of size NxN, input vector vec is size 1xN and the output vector product is of size 1xN. These same variable names are used in the figures below.

A. Loop Unrolling

In loop unrolling the body of the loop is replicated as many times as a given factor and the loop iterations are also decreased by that factor. So, the loop body does more computation than usual and all the statements in the body are executed in parallel if permitted by data dependency. The method provides the advantage of higher latency but at the cost of system size. This implementation creates the most parallelism possible [9. Loops can only be unrolled if the number of loop iterations are known beforehand. In the first optimized version of the code, only the inner loop was completely unrolled by the compiler. It should be noted that the inner loop iterations are dependent on each other, as the sum needs to be increased after each iteration. Figure 3 shows the implementation of a matrix vector program for 8 by 8 matrix. While this improved the latency, the resource utilization measured by the DSPs and LUTs was extremely high as compared to the basic kernel. The program did not even build for a matrix greater than 100 by 100 elements due to lack of space on the instance. A substantially better resource utilization was achieved by partially unrolling the inner loop by a factor of five by the compiler. On increasing this factor, the resource utilization becomes extremely high and the code cannot build due to lack of space on the current instance. To further optimize, the product variable was array partitioned. Local arrays are stored on BRAM resources by the FPGA, which is a dual port memory so only two elements can be written to the product vector in a cycle. By stating that product vector is kept in registers, multiple memory assignments can be performed in parallel, thus optimizing the program. Array partitioning is explained in subsection C. Two other versions were tested one with a partially unrolled outer loop along with a partially unrolled inner loop and another with a partially unrolled outer loop only. The first version required extremely large number of FPGA resources. However, the latter was efficient in terms of computational resources used but the latency was not optimized as much. In conclusion, a partially unrolled inner loop provides the most efficient results in terms of loop unrolling.

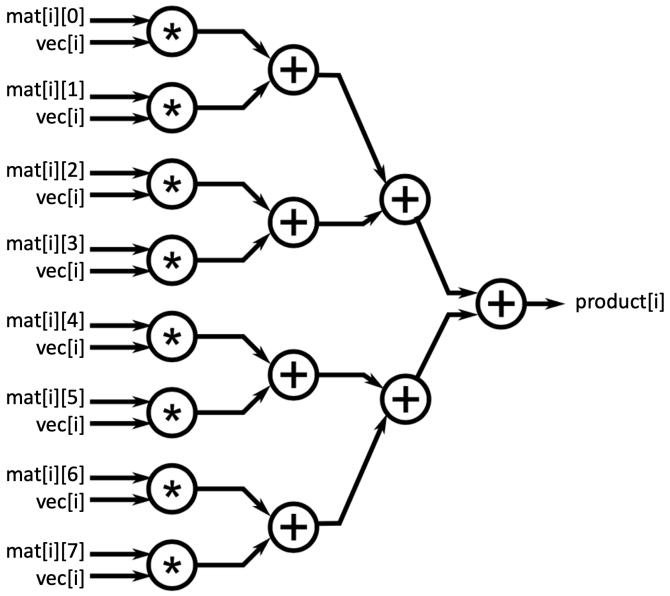

FIG. 3. The implementation of a single outer loop iteration of a completely unrolled inner loop for a matrix vector multiplication kernel for a 8 by 8 matrix.

B. Loop Pipelining

In loop pipelining, subsequent iterations of the loop body are pipelined. This means that different sections overlap and run concurrently instead of waiting for one loop iteration to finish executing. Initiation internal is the number of cycles between the start of two iterations of the loop. When a loop is pipelined in SDAccel, the compiler tries to achieve an initiation interval of 1, however this is not possible in our case due to the data dependency of the inner loop. Despite not achieving an initiation interval of 1 loop, pipelining significantly reduces the interval of the loop and does not affect the latency. Both versions with the inner loop pipelined and outer loop pipelined were executed and analyzed. When the outer loop is pipelined, the SDAccel compiler ties to flatten the nested loops and then pipeline them, however this is only possible if there are no data dependencies. Both the versions had similar number of DSP usage, which was significantly less than the unrolled loop, but

the latency optimization was minimal. Next, the outer loop was pipelined, and the inner loop was partially unrolled. This implementation used lesser computational resources than the unpipelined version. The pipelined behavior of the matrix vector multiplication product for an 8 by 8 matrix with a completely unrolled inner loop is shown in Figure 4. A further improvement would be to pipeline the multiplication operator used in the inner loop to allow a new multiplication to start every cycle with a certain latency. This will help reduce the overall loop latency. This implementation has not yet been tested. In conclusion, the most effective implementation involves pipelined outer loop with partially unrolled inner loop.

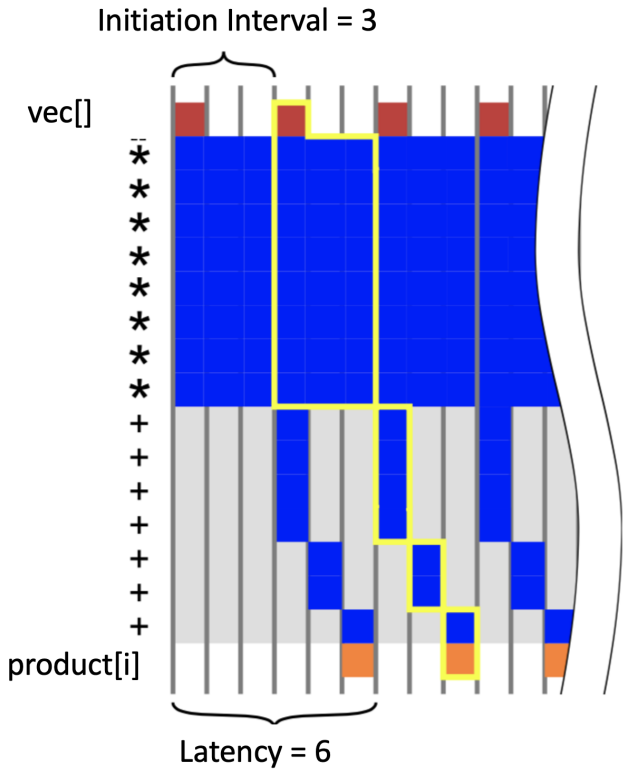

FIG. 4. The implementation of a single outer loop iteration of a completely unrolled inner loop for a matrix vector multiplication kernel for a 8 by 8 matrix.

C. Other Techniques

Array Partitioning is a technique which reorganize the data in an array into different array partition, each having its own memory ports. There are different techniques to partition the array - cyclic, block, complete. For accessing a matrix row wise for the matrix vector product, block partition is used where each partition is of row size. Complete partition, where each element is partitioned into a different array, was used in one version for both pipelining and loop unrolling. The advantage of doing this is allowing multiple access to the array during one cycle as compared to the dual memory port assigned to a single array. Data flow optimization is another method used to improve kernel performance. This refers to maximizing the usage of the available data bandwidth while transferring data between the global memory and the local memory.

Pipelined Inner Loop				
Size	LSI	LUT	DSP	FF
10	110	3466	40	5616
20	422	5888	80	10233
100	10015	1653	5	1898
1000	2000022	1925	5	1682
5000	5000022	1950	5	1699
Unrolled Completely Loop				
Size	LSI	LUT	DSP	FF
10	220	4879	100	6757
20	821	5123	200	10254
100	20022	32181	400	23940
1000	NB	NB	NB	NB
5000	NB	NB	NB	NB

The Table above shows the resultant computational resources and time for the kernel execution for various matrix sizes. LSI - Latency Start Interval - the amount of time that has to pass between invocations of a compute unit for a given kernel. FF, LUT, and DSP are variables used by the SDAccel Application to generate custom logic for each compute unit in the design. Similar tables were made for loop unrolling with a factor 2, factor 5, Size/2 number of work items and pipe lined outer loop for analysis. Apart from this the detailed kernel trace were also used for analysis.

III. CONCERNS AND FUTURE DEVELOPMENT

The next goal for this project would be to rewrite the Ising simulate code as a C++ kernel and host code. Designs choices will require choosing the parts of the code to optimize as kernel code and accelerated on the F1 instance and the remaining as host code which is run on the CPU. A number of different kernels will be used in this process, first being the optimized matrix vector multiplication kernel. A kernel which performs vector additions and another which scales vectors will be created using C++. These kernels will also be optimized using the methods described in Section III. Various versions of the Ising simulate goal will then be created using combinations of these kernels and the remaining code run by the host. When this was attempted in the past using OpenCL kernel, the program could not build and had strange use of computational resources. The code in the CIM GitHub repository for full FPGA is currently being used for this purpose. Once this code has been accelerated the most recent Ising simulator will be accelerated.

The long-term goal for this project, is to accelerate the MAX-k-SAT problem solver on the FPGA using the same process. Further progress will be made on this during the summer.

There have been a few issues regarding memory space on the F-1 instance. Performing a System Build and creating an Amazon FPGA Image (AFI) have not been possible for the optimized versions of the matrix vector multiplication code due to the lack of a powerful instance. Most of the System Estimate Reports used for analysis in Section III were based on the Hardware Emulation. Another difficulty was understanding the reason behind the behavior of the program during loop optimizations, especially the extremely high number of computational resources used during loop unrolling.

ACKNOWLEDGMENTS

I would like to thank Professor Peter McMahon for giving me the opportunity to work in his lab. I also want to thank Tatsuhiro Onodera for supervising me through this research project. I appreciate Marty Sullivan, from Cornell IT, for giving me access to use his AWS account to execute this project. I also want to thank Darren Schachter, who was previously working on this project, for guiding me through the whole process.

[1] P. L. McMahon, A. Marandi, Y. Haribara, R. Hamerly, C. Langrock, S. Tamate, T. Inagaki, H. Takesue, S. Utsunomiya, K. Aihara, and et al. A fully programmable 100-spin coherent ising machine with all-to-all connections. [Online]. Available: https://science.sciencemag.org/content/354/6312/614

[2] K. Schlachter and J. Tompson, "An introduction to the opencl programming model," Nov 2019. [Online]. Available: https://cims.nyu.edu/~schlacht/OpenCLModel.pdf

[3] "Opencl specification," *Opencl Specification - an overview — ScienceDirect Topics.* [Online]. Available: https://www.sciencedirect.com/topics/computer-science/opencl-specification

[4] "Fundamental concepts of application host code." [Online]. Available: https://www.xilinx.com/video/hardware/concepts-of-application-host-code.html

[5] 2009. [Online]. Available: https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/local.html

[6] "The opencl specification," 2009. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-1.0.pdf

[7] "Programming c/c++ kernels." [Online]. Available: https://www.xilinx.com/html_docs/xilinx2019_1/sdaccel_doc/rjk1519742919747.html

[8] "Kernel optimization." [Online]. Available: https://www.xilinx.com/html_docs/xilinx2019_1/sdaccel_doc/pxj1520531630838.html

[9] R. Kastner, J. Matai, and S. Neuendorffer, "Parallel Programming for FPGAs," *ArXiv e-prints*, May 2018.

[10] "Sdaccel development environment." [Online]. Available: https://www.xilinx.com/html_docs/xilinx2019_1/sdaccel_doc/uqp1519743299633.html\

[11] P. Frey, "Using sdaccel for host and accelerator code optimizations," Oct 2018. [Online]. Available: http://xilinx.eetrend.com/system/files/2019-01//private/100017045-56943-shi_yong_sdaccel_jin_xing_zhu_ji_ji_jia_su_qi_dai_ma_you_hua_.pdf

[12] S. Ghose and J. Tse, "Tuning the matrix multiply algorithm." [Online]. Available: https://jontse.com/courses/files/cornell/cs5220/project01_matmul.pdf

[13] "Kernel optimization: loop pipelining - other optimizations." [Online]. Available: https://zh-tw.coursera.org/lecture/fpga-sdaccel-theory/kernel-optimization-loop-pipelining-pj9Jz

[14] Xilinx, "Xilinx/sdaccel-tutorials." [Online]. Available: https://github.com/Xilinx/SDAccel-Tutorials/tree/master/docs/aws-getting-started/CPP

[15] D. Schachter, "Aws getting started with amazon instances." [Online]. Available: https://github.com/mcmahon-lab/CIMSimulator/blob/master/AWS/Getting\%20Started\%20with\%20Amazon\%20F1\%20Instances.pdf