A look into a GraphBLAS Entry Point into an LLVM Lowering Pass, with A Precision Formatting Example

Gulla, Roy
IEEE Northern Nevada Chapter
Mesquite Gaming, LLC
rgullape@gmail.com

***Abstract*— The Posits standard has shown itself to be well suited to high performance, and particularly, AI based processors as well as being a viable storage formatting alternative to IEEE754 float types. It contains an optional field in its format, the fractional bit field, which many times simply is bypassed if the number of storage bits does not allow for it. The purpose of the precision computing circuit presented here is to present a preprocessing approach to optimize processors for the storage space issues found with the implementation of the new formatting, and to show its compatibility to new offloaded memory array structures. The backbone of the compilation approach to instruction set optimization will be implementing single stage forwarding constraint, much as a carry flag is in traditional adder circuits, via a new computing pathway presented for the graphBLAS toolchain.**

Keywords—Number Representations•Rediuced Instruction Set Computer(RISC)• JIT Compilation• High Performance Graphics

I. INTRODUCTION

A large portion of the performance issues in accumulator based arithmetic circuits, and especially in grid based computing ones, lies in the address spacing needed in clustered machines. Some newer architectures are finding alternative computing pathways enabled by memory offloading controllers, and incorporating graphical processing engines.

In line with these newer architectures, the LLVM Compiler Infrastructure has produced a forked off CIRCT development project, which isolates the hardware lowering mechanism and ensures all development cycles are kept in sync with the LLVM compilation pass optimization scheme.

A recent but well known FPGA technique for edge computing ensures that sub 8 bit words are synchronously implemented in arithmetic instruction pipelines, shown in [1] to dramatically improve performance and reduce data loss, eliminating the need for such optimization techniques made popular in recent architectures as bit packing, data structure aligning, etc. In a promising development for implementing weighted computational pathways, single bit slices were implemented in single precision formatting for binary neural networks to incorporate fused multiply add accelerators. [1] This is just another example of where highly complex compute pathways are finding alternatives to pipelined stages (i.e. decode and execute) of floating point double precision numbers

These hardware advances, coupled with the recent advancement of John Gustafson's Posits format, should be exploited to fully incorporate both advancements together into recent efforts with IEEE standards updates, such as those with exact dot products.

The unique features of the Posits system which, while distinguishing it from IEEE 754 float, also leaves it prone to memory storage space issues, is the extra optional field for the fractional bits. In cases where it is not needed, the default value becomes zero, and the assembler is left with a non-instantiated extra value. So the fractional bit portion of the quire is left essentially dormant.

The intermediate stages of the pipelined circuit involving forwarding to eliminate redundant or wasted processor cycles can themselves create latency issues when cycles remain asynchronous, or when redundant and/or repeated load and stores are designed in the hardware. The emphasis here is not on changing any part of the execution stages, but on implementing an extra bit field in our circuit, the prefix field of our data operand. This is computationally the equivalent of an arithmetic micro operation of alignment (akin to the left shifting operator of an HLS circuit). By emphasizing a preprocessor compute structure, the circuit eliminates the creation of extra references in data storage, more specifically in the retrieval of these labels in the adder circuit.

The execution stage we wish to leave intact in order to exploit is the decoding of the exponent field, when all that needs to be known to start the pipeline is whether the exponent is under a certain value, (i.e. signed) and a single bit flag would suffice for this condition. Then the shadowing effect of our extra bit field would allow the decoding of the prefix of the current minposis value, and align the bits accordingly in order to encode the value of the subunitary number. What this latency in the arithmetic pipeline allows for is for logic-controlled data flows (i.e. usage of branch prediction tables), so we will be aligning our fractional bit field multiple times in a computational cycle. A more suitable option for our multiple (mantissa) alignments will be presented, once initial execution stage is completed, in order to minimize unnecessary spending of extra cycles with our innermost loop data and eliminate the need for rolling back the instruction part of the cache line. But certain tasks are better left to a LLVM based compiler's self loop optimization pass, and instead the attention of this paper turns to minimizing the

number of compiler passes spent prior to lowering to the hardware host.

TABLE 1A-B AN ENCODING OF A POSIT MINPOSIS FIELD OF BASE 4 AND EXPONENT OF -2

10	Prefix Field
	0
	1
	0
	0

a.

11	Prefix Field
	0
	0
	1
	1

‡The prefixes for the target machine are shown as a new proposed field in the POSITS formatting, with the sub-8 bit words shown shifted from the base position (at row one) to align the value in the adder circuit.

Fig. 1. The proposed encoder will be using something comparable to an indexed (weighted) crossbar matrix, with the individual columns shown above for an appropriate minposis exponent value, and it will load the prefix field weights in order to align the bit vectors of the matrix vector multiplication structure for the corresponding exponent bit field. In each presented half word, the topmost bit is the MSB, the bottom most is the LSB..

Cache lines for our sub 8 bit words will be overused in the case of the use of hardware buffers, and register memory at this stage of the pipeline, so as [4] shows to be a suitably secure alternative to creating unnecessary store (i.e. wait) cycles, the implementation at our stage of computation proposes the use of an appropriate open system intermediate buffer to hold the prefix 'weights' of the necessary summand terms of the current execution stage, much akin to the instruction memory of any pipelined ILP based machine.

While one of our block variables is in “wait” status for another instruction in the pipeline (exponent comparison branching) to execute (termed a shadow effect by the Intel microprocessor series), the word prefixes (which are the weights of any client implementing binary ANN structure) can be repeatedly decoded, thereby eliminating a RAW “hazard” in this stage of our pipeline.

// GB_jit__AxB_dot2__2c1f046bbb0bbbcd.c

```
//------------------------------------------------------------------------------
#include "GB_jit_kernel.h"

    // semiring: (plus, rdiv, double)
    // monoid:
        #define GB_Z_TYPE double
    #define GB_ADD(z,x,y) z = (x) + (y)
       #define GB_UPDATE(z,y) z += y
 #define GB_DECLARE_IDENTITY(z) double z =
                        0
#define GB_DECLARE_IDENTITY_CONST(z) const
                double z = 0
      #define GB_HAS_IDENTITY_BYTE 1
       #define GB_IDENTITY_BYTE 0x00
                   #define
   GB_PRAGMA_SIMD_REDUCTION_MONOID(z)\
      GB_PRAGMA_SIMD_REDUCTION (+,z)
       #define GB_Z_IGNORE_OVERFLOW 1
            #define GB_Z_NBITS 64
         #define GB_Z_ATOMIC_BITS 64
      #define GB_Z_HAS_ATOMIC_UPDATE 1
   #define GB_Z_HAS_OMP_ATOMIC_UPDATE 1
  #define GB_Z_HAS_CUDA_ATOMIC_BUILTIN 1
#define  GB_Z_CUDA_ATOMIC  CUDA_ATOMIC_TYPE
double
 // SuiteSparse:GraphBLAS v8.0.1, Timothy
         A. Davis, (c) 2017-2023,
          // All Rights Reserved.
  // SPDX-License-Identifier: Apache-2.0
 // The above copyright and license do not
                apply to any
    // user-defined types and operators
              defined below.
//------------------------------------------------------------------------------
                                    ----
```

Fig. 2. A look into a pragma based JIT framework approach: the SIMD vectorization pass in this case occurs at the graphBLAS CUDA Kernel level. The higher level pragma based branching mechanism (at implementation level) of our program will be shown in order to emphasize the need for only lowering to a sufficient level of non-generic structures, the main goal of the approach presented here.

II. MINIMIZING COMPILING PASSES WITH THE ACCELERATOR

One potential structure of the implementation of this is the offloaded multidimensional vectorized memory structure from such architectures. In a sense, the more open the system remains to cross-architecture compilation, the more capacity it has for a JIT compilation toolchain to implement it.

In the past bufferized memory blocks which not immediate, accessed by indirect branching can incur some performance penalties, but there is a new exception to this rule. Using the example of the programmable integrated unified memory architecture (PIUMA) by Intel, and implementing such a bufferized memory control structure, where GPU kernels could be fully implemented as the host device of choice, the approach then is to model the compute pathway in line with a CUDA centered matrix generating engine of choice, graphBLAS. The actual compiler based approach utilizes the necessary multi level intermediate representation (and the target source for lowering the compiler backend to is located in the dma buffer structure of Red Hat’s AMDGPU source file, and referenced in the code snippet here.)

```
/* decl.pdll*/
 (iprfx_, …)

#include ``amdgpu.h''
#include ``amdgpu_dma_buf.h''
#include ``amdgpu.h''
#include <drm/amdgpu_drm.h>
#include <linux/dma-buf.h>
#include <drm/drm_cache.h>

...
const struct dma_buf_ops amdgpu_dmabuf_ops
={\
  ..\
 .inc = iprfx_,
 \

};
```

Fig. 2. A dialect based typical llvm approach: the introduction of a pattern for a compilation backend tool is much preferable to introducing extra (and in substantial number of cases, unnecessary) labels into the assembler output.

In this above code exerpt from Red Hat's AMDGPU generic driver application, '.inc' is a boolean instruction type, since iprfx_ will indicate either an increase by the next available power of 2 in the particular regime's scale factor or a 0 value. But in this case the new intrinsic will need to be introduced into the entire compiler toolchain.

Any logic controller involved in the implementation will occur at block level. This will enable a customized compiler backend optimization approach, as opposed to reconstruction of the entire compiler tree. (As a particular file in the current llvm compiler pattern matching layer contains the defined 'prfx' constraint for preprocessing integer types, care should be taken to avoid bloating the generated instruction set with the addition of unnecessary constraints.)

In order to simplify the compiler toolchain system here, the option to create a compiler (or even a shell) flag is considered. With a single bit, the arithmetic circuitry can be localized down to a single node in the compiler toolchain by a point to point communication abstraction, implemented at the intermediate buffer of choice in multiple graphics processing card systems, dma-buf.

A simple check of the current LLVM "opsrun" python integrative test check shows that the necessary tensorizable implementations of arithmetic ops will provide the necessary hardware lowering, while minimizing extra vectorizing or other transformation passes, in order to streamline the tool and without extra phases in the compiler pipeline:

```
  .
   #opsrun.py
   linalg.fill ins(%v2 : f32) outs(%rhs :
memref<4xvector<8xbf16>>)
   linalg.fill ins(%v0 : f32) outs(%O0 :
memref<4xvector<8xf32>>)
   linalg.fill ins(%v0 : f32) outs(%O1 :
memref<4xvector<8xf32>>)
   call @elemwise_ipowi_on_buffers(%lhs, %rhs,
%O0) :
  (memref<f32>,memref<4xvector<8xbf16>>,memref
<4xvector<8xbf16>>) -> ()
```

Fig. 3. The pytorch tool's capacity for CUDA lowering transformations is mimicked and its streamlining of the compilation is highlighted here. However, as with all python lowering calls at hardware level, for instance with the bazel tool, sys needs to be invoked and so extra compiler framework security measures need to implemented.

In the above pytorch integrative approach both the bufferization as well as the fusion of ops occurs immediately prior to lowering to device machine levels, (or as is presented as a more viable AI suited approach for large scale systems in [6], outputting to a dma buffer) and these passes are condensed into the same node in the compiler tree.

A preprocessing directive set with CLI flags, or better, a basic compiler extension, would better allow for the host to construct a prefetched instruction buffer, in place of other more logic containing (prediction) buffers. Once the JIT compiler generated codes are assigned to the bufferized memory, repeated aligning (i.e. bit shifting) of the fractional bit field value would be much easier, as long as the clock delay cycle will allow it.

III. DIRECTIONS FOR FURTHER DEVELOPMENT

The following demo is shown of a cuda kernel driven JIT function suitable for an off-tree compilation of an LLVM compiler backend extension for our circuit:

```
/*GB_jit_launcher.cpp*/

#include "GB_jit_kernel.h"

      auto const& callthis=set_kernel_inst(GB_\
           _jit_kernel,mlir.MemRefType&\m,\
       dma_buf_PRIME& in_out);
       callthis(P->(_iprfx,),callthis(P->(in\
_out));/*tensorizable output */
```

Fig. 4. The above code is inspired largely by the Easy::JIT LLVM forked project from the 2018 European LLVM Developers Meeting, with appropriate alterations for the memory structures allocated for this implementation.

So here the suggested step is to utilize the graphBLAS toolchain as an alternative pathway to lowering to a LLVM's CIRCT node, in a way to maximize the immense block array memory optimizing capacities it demonstrates, and utilize its underlying matrix generation abilities. The project proposes its use as an off-tree JIT compilation backend tool, to complement NVIDIA's NVCC compiler tree. In presenting a fundamental pre-adder structure in this light, the optimal matrix based solution to the problem presents graphBLAS as an entirely fundamental tool in a lower level compilation framework.

The code referenced in this section of the paper can be viewed at https://gitlab.com/rgulla_nc.

ACKNOWLEDGMENT

The author wishes to extend his appreciation to Dr. John Gustafson and his team at National University of Singapore, as well as the 2023 Conference on Next Generation Arithmetic for their input and guidance in directing the course of this paper.

REFERENCES

[1] Bilaniuk, Wagner, Savaria, Jean-Pierre Bit Slicing FPGA Accelerator for Quantized Neural Networks (2019)

[2] H. Assaf, Y. Savaria, and M. Sawan, "Vector Matrix Multiplication Using Crossbar Arrays: A Comparative Analysis," 2018 25th IEEE International Conference on Electronics, Circuits, and Systems (ICECS), Bordeaux, France, 2018. pp609-612, doi: 101109/ICECS.2018.8617942.

[3] Guelton, Serge, Martinez Caamano, Juan Manuel. "Easy::JIT: compiler assisted library to enable just in-time compilation in C++ codes." Programming '18: Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming. April 2018 pp49-50, doi:10.1145/3191697.3191725

[4] Gustafson, J. (2015). The End of Error: Unum Computing. Boca Raton, Florida: CRC Press.

[5] Aananthakrishnan, S., Ahmed, N., Cave, V., Cintra, M.H., Demir, Y., Bois, K.D., Everman, S., Fryman, J.B., Ganey, I.B., Heirman, W., Hoppe, H., Howard, j., Hur, I., Koyiayath, M., Jain. S., Klowden, D., Landowski, M., Montigny, L., More, A., Ossowski, P., Pawlowski, R., Pepperling, N., Petrini, F., Sikora, M., Seshasayee, B., Smith, S., Szkoda, S., Tayal, S., Tithi, J.J., Vandriessche, Y., & Wrosz, I.P., (2020). PIUMA: Programmable Integrated Unified Memory Architecture

[6] H. Suh et al., "Algorithm-Hardware Co-Optimization for Energy-Efficient Drone Detection on Resource-Constrained FPGA," 2021 International Conference on Field-Programmable Technology (ICFPT), Auckland, New Zealand, 2021, pp. 1-9, doi: 10.1109/ICFPT52863.2021.9609840.

[7] Thorson, Gregory Michael. "Vector Computation Unit in a Neural Network Processor."United States Patent and Trademark Office. US20160342889A1. September 03, 2015