PROPOSAL: Sparse linear algebra hardware-software co-design

Aleksey Tyurin
Saint Petersburg State University
alekseytyurinspb@gmail.com

Daniil Berezun
JetBrains Research
daniil.berezun@jetbrains.com

Semyon Grigorev[*]
Saint Petersburg State University
s.v.grigoriev@spbu.ru

Abstract

In the era of big data, computations are expected to be faster and less power-consuming in order to become more effective and affordable. Sparse linear algebra is a great framework for building algorithms in a uniform and optimization-amenable way. However, CPUs and GPUs currently running such algorithms are underutilized due to being too general-purposed for problems that include sparsity. Thus, an application-specific integrated circuit could speed sparse computations up, following the example of *Google TPU's*. It is worth noting that such a circuit needs to be not self-contained to allow some expected optimizations to be made by a compiler or a framework itself. Finally, the optimizations should be easily definable in the language and as automated as possible, thus a careful simultaneous design of hardware and software is needed. The proposal describes the bottlenecks inherent to present sparse linear algebra framework implementations, summarizes the expected optimizations, and proposes a co-design approach to designing a highly-optimized sparse linear algebra framework. *This is a work in progress and not yet present the final result.*

Introduction

Sparse linear algebra is a great tool for tackling a wide range of data analysis problems and allows a definition of plenty of algorithms in terms of matrix and vector operations over some semiring for a huge area of applications including but not limited to graph analysis [?], computational biology [?] and machine learning [?]. *GraphBLAS* [?] standard defines building blocks, that could be implemented in software or hardware or both, for graph algorithms to be built in the language of linear algebra. A motivating example for a practical need of those blocks could be seen in listing 1, which is taken from a *Suite Sparse* talk[1]. The example performs breadth-first

```
#Sparse algebra BFS:
#times: if (A[i,j] != 0) A[i,j] × q(k) = k
#       else 0
#plus : any(x,y) = x or y randomly

q = [source];
parent = [0 for i in range(n)];
parent[source] = source;

while (q not empty)
    #masked matrix vector multiplication
    q<¬parent> = A*q #mask could be inverted
    #masked assignment
    parent<q> = q
```

Listing 1. Breadth-first search example

graph traversal using matrix-vector multiplication over a semiring. An operation could be parameterized by a mask, specifying what exactly elements are of particular interest: in case of a masked matrix-vector multiplication only masked elements are nonzero. Masked operations are essentially the operations followed by element-wise multiplication, thus element-wise operations are also a part of the standard. Notably, the multiplication operation of the semiring from the example returns the index of the element in the vector and the plus operation could nondeterministically return any of its parameters.

GraphBlas standard has been gradually being implemented for CPU and GPUs platforms with each release showing a performance enhancement. Following the work of [?], a number of straightforward optimizations could be emphasized that allow the implementation to catch up with the expected performance.

Direction-based optimization is in charge of either deciding to use sparse-vector or dense-vector operations for matrix-vector multiplication. The current frontier during a graph traversal could grow while the mask of yet not-visited vertices becomes small, so it becomes more profitable to utilize masked dense operations instead of sparse-matrix sparse-vector operations.

Load-balancing optimization chooses the most suitable distribution among the workers (e.g. threads) depending on the sparsity of the operands. Or schedules the operands in a way to save computations, e.g. first merge two small arrays

[*]Also with JetBrains Research.
[1]https://www.oden.utexas.edu/about/events/1520

Proposal, December, 2020

```
app [] ls = []
app (x:xs) ls = x : app xs ls

-- call for this function

app xs (app zs ys)

-- is fused to the following function definition
-- that is specialized for three lists

f [] xs ys = g xs ys
f (x:xs) ys zs = x : f xs ys zs

g [] xs = xs
g x:xs ys = x : g xs ys

-- and a call
f xs zs ys
```

Listing 2. Fusion of function composition

instead of a small and a large array in order to not extend a large array overhead throughout all the computation.

Mask fusion. Ahead-of-time masking could reduce the number of memory accesses in case of matrix-vector multiplication and prevent memory blow-up in case of matrix-matrix multiplication: the multiplication of two sparse matrices could produce an order of magnitude more nonzeroes in the output matrix compared with the two input matrices, hence the mask could limit the output number of nonzeroes, thus preventing out-of-memory errors. In order to achieve such a behavior, a mask should be fused (i.e. transformed in a single operation) with the corresponding operation for the operation to perform computations only for the elements in the mask.

Kernel fusion. Mask fusion is a special case of kernel fusion. Kernel fusion is responsible for fusing arbitrary operations. In the case of functional programming fusion [?] could be thought of as eliminating allocation of intermediate values of function composition, most notably for lists, since they are the primary data structure of functional programming. In the listing 2 append function `app` joins three lists, and fusion generates such code for function `f` that traverses each list only once. A simple motivating example for kernel fusion could be seen in listing 3. Masking, addition, and subtraction could be fused in one kernel code to prevent intermediate results allocation and redundant memory accesses for traversal for each or element-wise operation. The graph representation of operations could exploit operation properties, e.g., associative property, to perform the fusion. Further, the fusion for operations that could be represented as a composition of `map/reduce` operations are well-studied and could be effectively fused [? ?].

```
#Fusing together '+', '-', and masking
#prevents two intermediate matrices allocation

C<mask> = A + C - B
```

Listing 3. Kernel fusion example

Specialization. Generally, it is a program transformation optimization [?] that exploits the knowledge of some of the operation parameters, hence it could optimize those parts of the operation that depend on the known pieces of information. In the case of *NVIDIA CUDA* specialization assigns a specialized computation per a warp, since warps could be executed independently. This technique is well-described, e.g., in [?]. An example of such optimization could be a matrix-vector multiplication in some cycle where the vector remains unchanged. Hence, the vector could be embedded in the operation itself avoiding vector-specific overheads, e.g, allocation or data transfer back-and-forth from host to device. Matrix-vector multiplication is essentially a linear combination of columns where each column has a corresponding coefficient from the vector. Thus, each warp could multiply it's assigned column by the `warp_id's` element of the vector, which could be embedded in the code of the warp itself. Such embedding is generally a huge `warp_id switch` statement, which incurs almost no overhead in the case of CUDA.

However, there are inevitable challenges both for the hardware running sparse algorithms and for the software, performing optimizations.

For the former, both CPUs and GPUs remain underutilized when executing sparse operations due to high cache-miss rates, limited communication between processors [?]. Sparse algorithms are inherently memory-bound and thus results in poor GFLOPs number compared to peak GFLOPs theoretically available on modern devices, namely less than 0.2% of theoretical performance is achieved as reported in [? ?].

For the latter, optimizations are hard to automate and perform in general. The runtime of graph kernels is dependent on the input data, so in a multiple iteration algorithm, it might be profitable to fuse two kernels in one iteration and two different kernels in a different iteration. Further, kernels are compelled to satisfy certain restriction to be fuseable: the absence of intermediate synchronization, same data access pattern, enough resources on the device to execute fused kernel. Once again, `map/reduce` kernels could be successfully fused, but it is unclear whether GraphBlas standard could be solely implemented in `map/reduce` terms. For specialization, it is better for the underlying hardware to be *MIMD* for the workers to be completely independent, however GPU's architecture is *SIMT* (or *SPMT*), which prevents successful specialization in many cases.

Eventually, some application-specific integrated circuits have been designed to address the issues mentioned above, that basically provide hardware units for sparse matrix-matrix or matrix-vector multiplication [**? ? ? ?**]. A brief overview could be found in [**?**]. The implementation from [**?**] greatly outperforms CPUs (Intel MKL[2]) and GPUs (cuSPARSE[3]) solutions in terms of speed and power consumption. Despite high performance, such solutions do not yet provide a complete implementation of GraphBlas (or even a subset required for a concise BFS from the example) and seem to be too self-contained to split the operation into phases that could be optimized in the discussed sense.

Thus, a solution that combines the hardware and software in co-design fashion considering both hardware and software bottlenecks could be proposed. The hardware should be general enough to allow software optimizations and the optimizations should be easily definable in the co-designed language. A possible approach to tackle the design problem is to use a co-design framework [4] and TTA processor architecture, which is highly parallel and customizable (allowing, e.g., MIMD computations).

Currently, sparse algebra frameworks speed up graph databases, e.g., RedisGraph [5], computational biology and machine learning, and equipping cloud solutions that provide such services with the dedicated sparse hardware could both increase the performance of the queries and make the services more affordable by reducing power consumption following Google's TPUs [6] example.

1 Problem statement

LA but whole program optimization. Fusion and so on.

Static vs dynamic data. Specialization requires MIMD.

Representation for data (formats)

2 Sparse LA arch

3 Graph processors

4 Partial Evaluation

Partial cases for different models (array programming, stream fusion)

For functional languages.

5 Lambda processors

6 Dataflow processors

TTA and other dataflow + MIMD

7 Roadmap

Library + language + compiler + hardware codesign.

[2]https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html
[3]https://developer.nvidia.com/cusparse
[4]http://openasip.org/
[5]https://oss.redislabs.com/redisgraph/
[6]https://cloud.google.com/tpu