Whether and How to In-Cache Compute

Alireza Khadem
arkhadem@umich.edu

Ashutosh Mishra
ashumich@umich.edu

Steven Schaefer
stschaef@umich.edu

Sumanth Umesh
sumanthu@umich.edu

***Abstract*—In-cache computing exploits the massive parallelism of SRAM array bit-lines to do in-situ computing. Although existing literature provides both domain-specific and general-purpose in-cache computing designs, it lacks a proper compilation framework. Specifically, one needs to know *Whether* and *How* an application should be offloaded to the in-cache computing engine. In this work, we answer these questions by implementing an LLVM pass that analyzes scalar code based on multiple schedules. Our experimental results show that our project is able to predict the in-cache computing cycles for a variety of data-parallel applications with 86% accuracy.**

I. Introduction

In-cache computing converts the bit-lines of cache SRAM arrays to ALUs. By activating multiple word-lines simultaneously, sense amplifiers compute AND and NOR of two word-lines. Prior works [3] add extra logic to the SRAM array peripheral to compute more complex operations in-situ in the cache. Using in-cache computing is beneficial as: (a) it removes the data transfer between core and cache, resolving the well-known *Memory Wall*, and (b) exploits the massive parallelism of cache lines for efficient computing.

In-cache computing architecture is extensively explored for both domain-specific and general-purpose applications. Neural Cache [6] exploits the massive parallelism of in-cache SIMD lanes for convolutional neural networks. Duality Cache [7] uses Single-Instruction Multiple-Threads (SIMT) execution model to map the CUDA kernels to the bit-lines, which is beneficial for both task and data-parallel applications. Compute Caches [3] exploits Single-Instruction Multiple-Data (SIMD) execution model to treat in-cache computing engine as a vector processor. Our work is an extension of this model.

Despite the extensive research on the architecture and system-level problems of in-cache computing, it lacks a general-purpose compilation infrastructure. Since this programming paradigm is new to the computer systems community, one might wonder *which applications are suitable for in-cache computing and how these kernels ought to exploit the inherent parallelism of data-parallel applications.* In this work, we try to answer these two questions using compiler analysis.

Although these questions are studied in prior work [4] for Processing In Memory (PIM), we believe that PIM has extensive differences with in-cache computing. First, general-purpose cores of PIM are able to execute irregular control flow of task-parallel applications, yet, SIMD lanes of in-cache computing can only benefit from the regular control flow of data-parallel kernels. Second, cost modeling of these two processors are different. A bank-level PIM architecture requires reordering addresses between banks of a DRAM to provide local access to private data for each bank-level processor. On the other hand, in-cache computing uses vector memory accesses to load/store data from/to the memory. Finally, host (CPU) - device (DRAM) communication of the PIM engine is coarse-grain, meaning that both scalar and vector operations are executed locally in the bank-level general-purpose processors. However, our execution model executes scalar operations on the host (CPU) and offloads vector operations to the device (Cache).

Thus, we observe that in-cache computing requires a general-purpose compilation infrastructure to make this programming model easier for programmers. Our work takes the burden of whether and how to offload to an in-cache computing engine from the shoulders of programmers, unleashing the power of in-cache computing for data-parallel applications.

II. Background

A. In-Cache Computing Engine

For an SRAM array to operate on two data elements, prior work reorganizes the data from a horizontal to a vertical layout [3], [6]. Complex operations are converted to simple micro-operations that are executed bit-serially in the cache. Figure 2 shows this data layout where data elements of an array are located vertically in separate bit-lines (BL) of an SRAM array (SA). Therefore, in-cache computing provides $\#SA \times \#BL/SA$ SIMD lanes as a bit-serial vector engine. For example, by dedicating half of the L2 cache of the ARM Cortex-A76 Prime processor for in-cache computing, one can operate on $32 * 256 = 8192$ data elements in parallel.

B. Polyhedral Compilation

Polyhedral compilation [2] models nested loop iterations as representing integer lattice points of a polyhedron. Using this point of view, reasoning on the structure of these polyhedra can reveal latent optimizations and parallelism that are not readily apparent when looking at the original presentation of the loop.

A transformation is an operation on loops that scales, swaps, or adds constants to loop iterables $\{i, j, k, \dots\}$. Some examples of transformations include loop reversal — i.e. instead of $i = 0, i \leq N, i := i + 1$, we go in the opposite order $i = N, i \geq 0, i := i - 1$ — and loop interchange — i.e. if i corresponds to the outer loop and j to the inner loop, instead we will reconfigure the code to make j the outer loop and i the inner. More generally, a loop transformation is valid if it respects the cross-iteration data dependencies of the original loop and is affine with respect to loop iterables and loop invariants. Informally, this says that we may treat a loop like a set of instructions (indexed by the values of $\{i, j, k, \dots\}$) that may be executed in any order that respects dependence constraints.

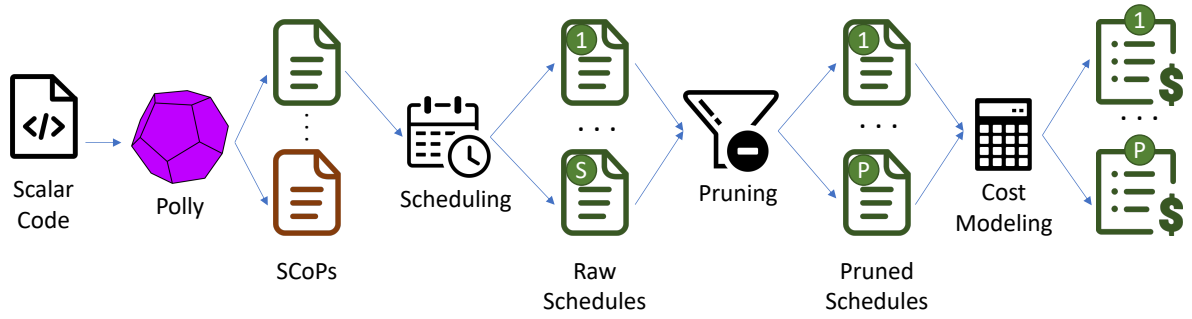

Fig. 1. The proposed **Compilation Pipeline** takes scalar code as input. We use a *Polyhedral Compilation* [1] framework to detect the vectorizable SCoPs, and *Loop Transformations* [8] to produce multiple schedules of a nested loop. We prune the schedules and remove reductions. Cost modeling analyzes the code and provides a cost for each schedule. The cheapest schedule is chosen for a SCoP.

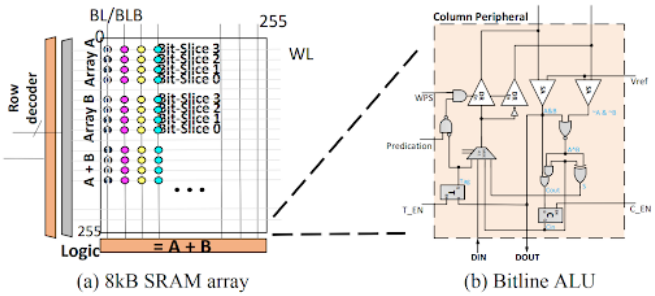

Fig. 2. SRAM modifications for in-cache computing. Figure from Neural Cache paper [6].

The polyhedral model is precisely the method by which we reason about and reorder this set of instructions.

We make use of Polly [1], a polyhedral compilation library built into the LLVM project. Further, much of Polly's interface obfuscates how to manually transform loops, as the primary use of the library is for finding optimizations, which it automatically searches for, rather than allowing a user fine-grained control of their loops. Previous work [8] has inserted some directives as a wrapper around Polly to easily allow users to apply specific loops transformations. Later, when referring to transformations that we have applied to a loop, we are talking about this process of user-directed loops via compiler directives.

III. Compiler Design

A. Assumptions

In the interest of time, we make some assumptions about the input programs as well as the in-cache computing engine. In our evaluation, we take our inputted loops to have constant bounds and increment the loop iterable by 1 each iteration. Certainly, not all loops fall into these categories; however, in many cases where there the loop increments by 2, or has variable bounds, the loop could be made to have constant bounds and unit increment if refactored. That is, in either presentation, each program would refer to the same iteration polyhedron, so for the purposes of this work we can consider them equivalent.

Our search space for loop transformations is only interchanges (including associated clean up code). It could be fruitful to later expand this search into other transformations. It is legal to transform a loop acting on the loop schedule by any affine transformation. Because there are infinitely many affine transformations, adding even one other class of transformations to our search space has the potential of causing a computationally infeasible blow-up.

Likewise, we don't consider cache under-utilization due to complexity. Finally, we also make use of simple terms in our cost model for computing in the cache. We distinguish only between memory instructions and non-memory instructions. Not every arithmetic instruction is guaranteed to have the same latency, but the core logic for computing their execution time is the same. In future work, this may be an interesting aspect to describe in finer detail.

B. Compilation Pipeline

This section explains our compilation pipeline, which is presented in Fig. 1. It takes in scalar code in LLVM Intermediate Representation (IR) and provides a cost for each scheduling of all SCoPs.

Static Control Parts (SCoPs). LLVM IR is analyzed by Polly and split in Static Control Parts (SCoPs). Each SCoP is a nest of loops that shares static control flow. That is, if any part of the nest is executed, then all of the nest is executed. The SCoP is the atomic building block of a polyhedral analysis of the code, so each SCoP is analyzed independently. This means that our system analyzes a single SCoP at a time.

Loop Schedules. A program's *schedule* is the traversal order through the iteration polyhedron. When applying a loop transformation, we are editing its schedule. For each SCoP, we aim to find a schedule that would elucidate some latent vectorizability that was not apparent in the original presentation of the loop. To this end, we fix a class of loop transformations to act as our search space: loop interchanges. To mitigate the complexity of this project, we only vectorize over the innermost loop. That is, when generating schedules, if given m nested loops we iterate over all m different choices of innermost loop. Given sufficient time, it may be useful to enhance this project with a more thorough vectorization procedure so that in-cache resources may be utilized to their fullest potential.

Schedule Pruning. If a schedule is determined to be not vectorizable, we simply toss it out of our search space. The

largest roadblock to vertorizability occurs when the innermost loops is a *reduction*. A reduction is a loop that combines the output of a multiple parallel computations into a single output. These types of operations are well-suited for being run in SIMD, rather they are the meant to recombine data after SIMD has been executed. If we tried to run a reduction on the in-cache computing engine, the runtime would be too high due to the expensive cost of access memory from within the cache. By looking at the IR of the scalar code, we may determine if a loop is a reduction or not. For instance, if the definitions generated within the loop body are only used by a single operation in the postheader, then the loop is likely a reduction. Again, we use methods from Polly in our system to achieve this reduction detection.

Detecting Vector Instructions. Polly provides a list of loads and stores that will be vectorized. However, we do not know which non-memory instructions (such as `add`, `mul`, `sub`) need to be vectorized. Algorithm 1 describes a backtracking procedure that starts at a non-scalar store in LLVM IR and iteratively traverses the use-def chain until finding a corresponding non-scalar load. All instructions that are traversed in between the procedure are categorized as vectorized. We ignore instructions such as `getelementptr` and `phi` since their latencies are already accounted for by other instructions in the SIMD architecture.

Algorithm 1 Vector Operation Detection

```
VecOPs = []
for VecST in Stores do
    if VecST is Scalar then
        Continue
    else
        Frontier = [VecST]
        while Frontier not Empty do
            CurrOP = Frontier.pop()
            VecOPs.push(CurrOP)
            for UseOP in CurrOP.Uses() do
                if UseOP not Load then
                    Frontier.push(UseOP)
                end if
            end for
        end while
    end if
end for
return VecOPs
```

Cost Modeling. Once we have compiled vector code with respect to a given schedule, we now estimate how long it would take that code to run on the in-cache computing engine. We may compute the execution time of the scalar code on the CPU in the same way as [5]. To calculate the in-cache runtime estimate we need to find the number of times each vector instruction will be run on each the SIMD lanes.

Consider the code in Listing 1, the instruction `sum += A[j][i]` will be executed a total of 64×32 times. Similarly, `x = B[k][i]` will be run $128 \times 64 \times 32$ times. We divide this by the vector width (number of SIMD lanes) to get the value of *Count*. In addition, for memory operations, we need to find the number of cache lines that will be used, which is based on the *stride* — change in address for a memory access across one iteration of the innermost loop. In Listing 1 `x = B[k][i]` has a stride of 128. This is because for each increment of k we go from $B[k][i]$ to $B[k+1][i]$. The difference between these two locations is equal to the size of the dimension represented by i which is 128.

```
// A-> float[1024][256]
// B-> float[512][128]
for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 64; j++) {
        sum += A[j][i];
        for(int k = 0; k < 128; k++) {
            x = B[k][i];
        }
    }
}
```

Listing 1. SCoP Example

Once we have the *Stride*, we can compute the number of cache lines needed using the following equation

$$\#CacheLines = \frac{Stride \times VectorWidth \times BytesPerElement}{BytesPerCacheLine}$$

The total cost is computed as a weighted sum between the instruction *Count* and the instruction latency shown in Algorithm 2.

Algorithm 2 Cost Calculation

```
Cost = 0
for VecOP in VecOPs do
    Count = CalcDomain(VecOP) / 8192
    switch (type(VecOP)
        case LD/ST: Cost += CalcCacheLine(VecOP) × MemAccLatency × Count
        case COMP: Cost += CompLatency(VecOP) × Count
    end switch
end for
return Cost
```

Since we run our analysis pass on scalar code, there are instances where we end up with more vectorized memory accesses than required because very large arrays are assumed to spill to memory while it is not necessary in our case as we have 8 vector registers of width 8192×32 bits. We implement checks in our pass that ensures such extra memory operations are not considered for cost calculation.

Once it has made these estimates, the compiler then either tags the SCoP to be run on the CPU or offloads the vectorized code (configured with the fastest schedule to the in-cache computation engine), whichever is the cheapest.

IV. Experimental Methodology and Results

We use a prior cycle-accurate simulator to estimate the execution time of data-parallel applications. These simulator models ARM Cortex-A76 Prime processor. We dedicate half of the L2 cache for in-cache computing and the other half for storage.

We use a variety of data-parallel applications to compare the estimated cost with the simulation results. We choose General Matrix Multiplication (GEMM), Finite Impulse Response (FIR)

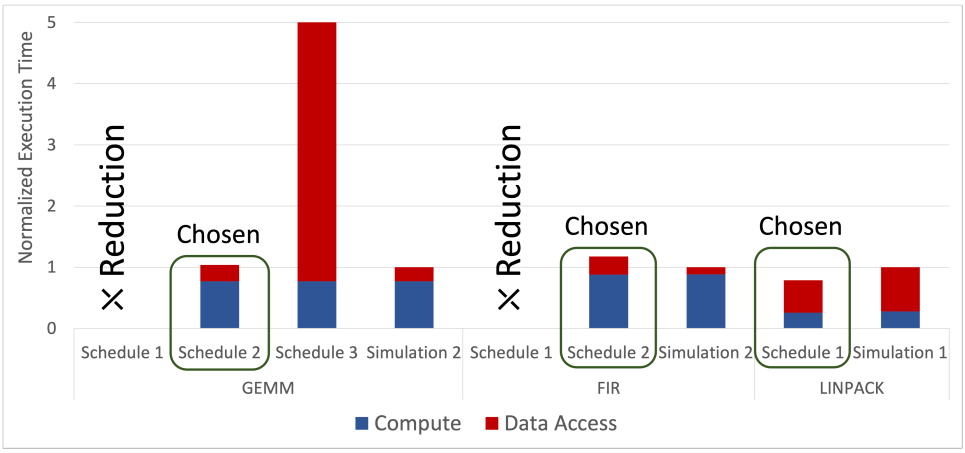

Fig. 3. Stimated execution time of different schedules normalized to the simulation time. Some schedules are ignored due to the reduction operations.

filter from Digital Signal Processing, and Linpack kernel (LINPACK) for linear algebra. For each of the kernels, we generate N schedules such that each loop occupies the position of the innermost loop at least once, where N is the number of nested loops in a SCoP. Next, we estimate the cost of each schedule and simulate only the schedule with the lowest cost.

We have 3 schedules for the GEMM kernel, 2 schedules for the FIR kernel, and 1 schedule for LINPACK kernel. Out of these 1 GEMM and 1 FIR kernel are *pruned*. These schedules correspond to a reduction operation in the innermost loop, which cannot be vectorized. Fig. 3 shows the estimated runtime for the different schedules as well as the actual simulated value. On average, our estimated runtime for the best schedule is 86% accurate.

For GEMM, schedule 3 has a very long data access time compared to both schedule 2 and the simulation result. This is because data access in this schedule is column-wise whereas the data is stored in a row-major format. This strided access leads to a much larger number of memory accesses.

For FIR and LINPACK, there is a small difference between the estimated data access time and the simulated data access time. We speculate that this is because our cost model uses an average value for the memory access latency whereas the kernel has a different cache hit rate in simulation.

A. Future Work

Our work here provides an interesting and adequate estimation of the in-cache run time; however, these aspects merit further investigation:

1) Generating loop schedules with more transformations than just interchanges.
2) Generalize loop bounds from only constants to affine expressions in the loop iterables and loop invariants.
3) Investigate the impact of under-utilized SIMD lanes, and mitigate the opportunity for under utilization to occur by vectorizing across multiple loops.
4) Make the cost model more expressive for each of the computation instructions, rather than a single term representing all non-memory instructions.

V. Conclusion

This project has opened the door for further investigation into optimizing resource usage for in-cache processors. We have found that for a small suite of benchmarks, we can produce a list of vectorizable schedules, and for each of these schedules we can accurately estimate the execution time of vectorized code when run in-cache. This analysis imbues the compiler with the static knowledge needed to best utilize the resources available to it.

References

[1] "Polly: Llvm framework for high-level loop and data-locality optimizations," https://polly.llvm.org/, accessed: 2022-11-21.

[2] "Polyhedral compilation," https://polyhedral.info/, accessed: 2022-11-21.

[3] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 481–492.

[4] A. Devic, S. B. Rai, A. Sivasubramaniam, A. Akel, S. Eilert, and J. Eno, "To pim or not for emerging general purpose processing in ddr memory systems," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 231–244. [Online]. Available: https://doi.org/10.1145/3470496.3527431

[5] A. Devic, S. B. Rai, A. Sivasubramaniam, A. Akel, S. Eilert, and J. Eno, "To pim or not for emerging general purpose processing in ddr memory systems," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 231–244. [Online]. Available: https://doi.org/10.1145/3470496.3527431

[6] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, p. 383–396. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00040

[7] D. Fujiki, S. Mahlke, and R. Das, "Duality cache for data parallel acceleration," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 397–410. [Online]. Available: https://doi.org/10.1145/3307650.3322257

[8] M. Kruse and H. Finkel, "User-directed loop-transformations in clang," in *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, 2018, pp. 49–58.