EXPLORING HW/SW CODESIGN TO ACCELERATE CFD APPLICATIONS

EE 593: Dual Degree Project 1

Presenter: Rishabh Ravi
Guide: Prof. Virendra Singh

Computer Architecture & Dependable Systems Laboratory,
Department of Electrical Engineering, IIT Bombay

INTRODUCTION

- Computational Fluid Dynamics (CFD) applications are a popular category of applications, that are known to be computationally heavy
- They constitute 25% of SPEC 2017 benchmarks, and 33% of SPEC 2006 benchmarks

Some common CFD tools

BACKGROUND

- Computational fluid dynamics is a branch of fluid mechanics that uses numerical analysis and data structures to analyze and solve problems that involve fluid flows
- There are three parts of a computational fluid dynamics (CFD) tool
  - Pre-Processing: Users can build a model using the CFD tool's modeling module or professional modeling software like CAD or UG.
  - Solver: Users can choose different solvers to conduct calculations.
  - Post-processing: The final step in the CFD workflow.

BACKGROUND

- CFDs are usually slow, characterized by high memory utilizations and parallelizable code
- Predominantly matrix manipulations, with the matrices being sparse
- To further identify and understand potential bottlenecks we begin to profile CFD tools

PROFILING TOOLS

Perf

- a simple command line interface based on the perf_events interface exported by recent versions of the Linux kernel
- Captures the amount of time spent in a specific function

Valgrind

- tool that help you make your programs faster and more correct.
- captures the number of exclusive instructions utilized for each functions

LIST OF PROFILED APPLICATIONS

Application	Description
SPEC 2017 lbm	This program implements the so-called "Lattice Boltzmann Method" (LBM) to simulate incompressible fluids in 3D as described
SPEC 2007 dealii	The testcase for the benchmark version, 447.dealII, solves an equation (a Helmholtz-type equation with non-constant coefficients) that is at the heart of solvers for a wide variety of applications.
OpenFoam	While primarily a computational fluid dynamics (CFD) tool, OpenFOAM includes FEM solvers for structural mechanics and other types of problems
ReCoDE-SPH-solver-2D-NS	A numerical code in C++ which solves the two-dimensional Navier-Stokes equations using the smoothed-particle hydrodynamics (SPH) approach.
SpectralDNS	spectralDNS contains a classical high-performance pseudo-spectral Navier-Stokes DNS solver for triply periodic domains.

APPLICATION PROFILE

INFERRING PROFILING RESULTS

- Slow code detected by *valgrind* is one that has the greatest number of instructions
- *Perf* detects the part of code that takes the most time to execute
- This could point to functions that have unpredictable branches, excessive loops, or memory bottlenecks

CHARACTERIZING APPLICATIONS

- We characterize functions by finding their *Throughput* given as Instructions/Execution time (GIPS)
- A better metric would be Flops but as we are measuring the throughput of functions, they are proportional
- *Throughput* is used to identify if functions are bottlenecks, by comparing against maximum throughput of the system
- The maximum throughput for an eight-core system operating at 4.2GHz is 50.4 GIPS

APPLICATION PROFILE

Application	Function	Ins count	Execution Time	Throughput
SPEC 2017 lbm	PerformStreamCollide	98%	97.25%	7.088
SPEC 2007 dealii	Compute_fill	-	23.90%	-
OpenFoam	Smooth	47.73%	45.77%	9.09
	residual	25.99%	30.05%	10.965
ReCoDE-SPH-solver-2D-NS	NeighbourParticlesSearch	9.28%	18.07%	4.747
SpectralDNS	Main	12.99%	27.79%	2.58

- With the bottlenecks identified, we dive deeper into the code

SPEC 2017 LBM

- The code of lbm shows that the bottleneck arises due large code size, most of which can parallelized
- The code performs multiple operation of the form

```
DST_N[dstGrid] = (1.0-a)*SRC_N[srcGrid] + b + c;
```

- Although not a bottleneck, it has scope to improve by exploiting parallelizability

Function	Data cache reads	L1 cache read misses	L3 cache read misses	Data cache writes	L1 cache write misses	L3 cache write misses
PerformStreamCollide	97.3%	95.13%	95.15%	97.6%	99.6%	97.8%

SPEC 2017 DEALII

- The code of dealii once again shows that the bottleneck arises due large code size, most of which can parallelized
- The code performs multiple operation of the form

```
c[point] += b* a[k];
```

OPENFOAM

- The bottlenecks in these applications are functions that work on sparse matrices – GaussSiedelSmoother::smooth, ldumatrix::residue
- The sparse matrices are either diagonal, upper triangular or lower triangular matrices

OPENFOAM

- The GaussSiedelSmoother function iteratively solves the linear equation

$$Ax = B \Rightarrow \quad Lx[n] = B - Ux[n-1] \quad (A = L+U)$$

- The current algorithm uses a loop to evaluate x[n] one row at a time
- A few methods to optimize would be to introduce a new way to perform GaussSiedelSmoother, one that does it in the hardware

[1] Massimiliano Culpo, 'Current Bottlenecks in the Scalability of OpenFOAM on Massively Parallel Clusters'

RECODE-SPH-SOLVER-2D-NS

- The SPH solver has the following statistics
- NeighbourParticleSearch has multiple loops of the following type

```
for (size_t i = 0; i < numberOfCells; i++) {
    for (size_t j = 0; j < cells[i].size(); j++) {
      for (size_t k = 0; k < cells[i].size(); k++) {
            distanceX = a[i]b[j] + a[i][k];
            c[i][j] = a[i][k];
      }
   }
}
```

SPECTRALDNS

- SpectralDNS only has a main function, so we analyse which code segment takes most time

```
U_tmp[z] = V[z]*CW[z]-W[z]*CV[z];
V_tmp[z] = W[z]*CU[z]-U[z]*CW[z];
W_tmp[z] = U[z]*CV[z]-V[z]*CU[z];
```

- These three lines contain 27% of the total instruction, 38% of the total reads from data cache, 21% of total writes to data cache
- It accounts for 40% of the total last level cache accesses
- The remainder of the code is also of a similar type, with instructions in large loops

MOTIVATION

- The bottlenecks are different forms of matrix operations
    - `c[point] += b* a[k]`
    - `U_tmp[z] = V[z]*CW[z]-W[z]*CV[z]`
- We now explore the benefits of having hardware and software acceleration and proceed to build them
- As the bottlenecks are matrix operations, they are the ideal candidates for prefetching and multithreading

SOFTWARE PREFECTHING

SOFTWARE PREFETCHING

- The programmer or compiler inserts prefetch instructions into the program
- These are instructions that initiate a load of a cache line into the cache, but do not stall waiting for the data to arrive
- A type of software prefetching is explicit software prefetching, where the developer uses prefetching primitives to prefetch the data
- On GCC and CLANG, prefetching is done using __builtin_prefetch builtin

SOFTWARE PREFETCHING

```
SWEEP_START( 0, 0, 0, 0, 0, SIZE_Z )
    __builtin_prefetch(&srcGrid[C + i + N_CELL_ENTRIES]);
    __builtin_prefetch(&srcGrid[N + i + N_CELL_ENTRIES]);
    __builtin_prefetch(&srcGrid[S + i + N_CELL_ENTRIES]);
    __builtin_prefetch(&srcGrid[E + i + N_CELL_ENTRIES]);
    __builtin_prefetch(&srcGrid[W + i + N_CELL_ENTRIES]);
    __builtin_prefetch(&srcGrid[T + i + N_CELL_ENTRIES]);
    __builtin_prefetch(&srcGrid[B + i + N_CELL_ENTRIES]);
    __builtin_prefetch(&srcGrid[NE + i + N_CELL_ENTRIES]);
    __builtin_prefetch(&srcGrid[NW + i + N_CELL_ENTRIES]);
    __builtin_prefetch(&srcGrid[SE + i + N_CELL_ENTRIES]);
    __builtin_prefetch(&srcGrid[SW + i + N_CELL_ENTRIES]);
    __builtin_prefetch(&srcGrid[NT + i + N_CELL_ENTRIES]);
    __builtin_prefetch(&srcGrid[NB + i + N_CELL_ENTRIES]);
    __builtin_prefetch(&srcGrid[ST + i + N_CELL_ENTRIES]);
    __builtin_prefetch(&srcGrid[SB + i + N_CELL_ENTRIES]);
    __builtin_prefetch(&srcGrid[ET + i + N_CELL_ENTRIES]);
    __builtin_prefetch(&srcGrid[WT + i + N_CELL_ENTRIES]);
    __builtin_prefetch(&srcGrid[EB + i + N_CELL_ENTRIES]);
    __builtin_prefetch(&srcGrid[WB + i + N_CELL_ENTRIES]);

    if( TEST_FLAG_SWEEP( srcGrid, OBSTACLE )) {
        DST_C ( dstGrid ) = SRC_C ( srcGrid );
        DST_S ( dstGrid ) = SRC_N ( srcGrid );
        DST_N ( dstGrid ) = SRC_S ( srcGrid );
        DST_W ( dstGrid ) = SRC_E ( srcGrid );
        DST_E ( dstGrid ) = SRC_W ( srcGrid );
        DST_B ( dstGrid ) = SRC_T ( srcGrid );
        DST_T ( dstGrid ) = SRC_B ( srcGrid );
```

Prefetching in LBM

```
if (update_flags & update_q_points)
  for (unsigned int point=0; point<n_q_points; ++point){
    for (unsigned int k=0; k<data.n_shape_functions-7; ++k){
      __builtin_prefetch(&data.mapping_support_points[k+7]);
      quadrature_points[point]
        += data.shape(point+data_set,k)
           * data.mapping_support_points[k];
    }
    for (unsigned int k=data.n_shape_functions-7; k<data.n_shape_functions; ++k){
      quadrature_points[point]
        += data.shape(point+data_set,k)
           * data.mapping_support_points[k];
    }
  }
```

Prefetching in Dealii

OBSERVATIONS

- Software prefetching has shown to be beneficial to these applications

Application	Speedup	Reason
SPEC 2017 lbm	1.024	By prefetching we avoid stall caused by a cache miss, happening a large number of times
SPEC 2007 dealii	1.075	Achieved speedup of 1.075629573
ReCoDE-SPH-solver-2D-NS	1.001	repeat 10000(repeat 4(repeat 4(…))) Each block (4B) accessed gets one miss every 4 iterations of outer loop, but hits every other time, so no significant speedup

OBSERVATIONS

- In all cases, strided prefetching helped improved performance
- This highlights the nature of the code present in CFD applications
- The stride length was adjusted based on the size of the data being fetched
- Ensuring the data was available before being accessed allowed us to avoid the stalls caused due to cache misses

HARDWARE PREFECTHING

HARDWARE PREFETCHING

- We compare the results after running these application on the same system with the prefetchers disabled

Application	Speedup
SPEC 2017 lbm	2.31
SPEC 2007 dealii	1.292
OpenFoam	1.05
ReCoDE-SPH-solver-2D-NS	1.125
SpectralDNS	1.607

MULTITHREADING

MULTITHREADING

Application	Speedup for number of threads				
	1	2	4	5	10
SPEC 2017 lbm	1	1.4	1.17	1.13	1.12
SPEC 2007 dealii	1	1.03	1.02	1.02	1.02
ReCoDE-SPH-solver-2D-NS	1	1.01	0.99	1.05	0.78
SpectralDNS	-	-	-	-	-

- OpenFoam does not allow modifications, so multithreading was not possible
- SpectralDNS uses MPI_FFTW library already

OBSERVATIONS

- The speedup achieved by hardware prefetching suggests that data movement is one bottleneck
- The speedup from multithreading shows the scope of parallelizability of code

POSSIBLE SOLUTION

- Instead of moving data to the CPU, we could move the execution towards data

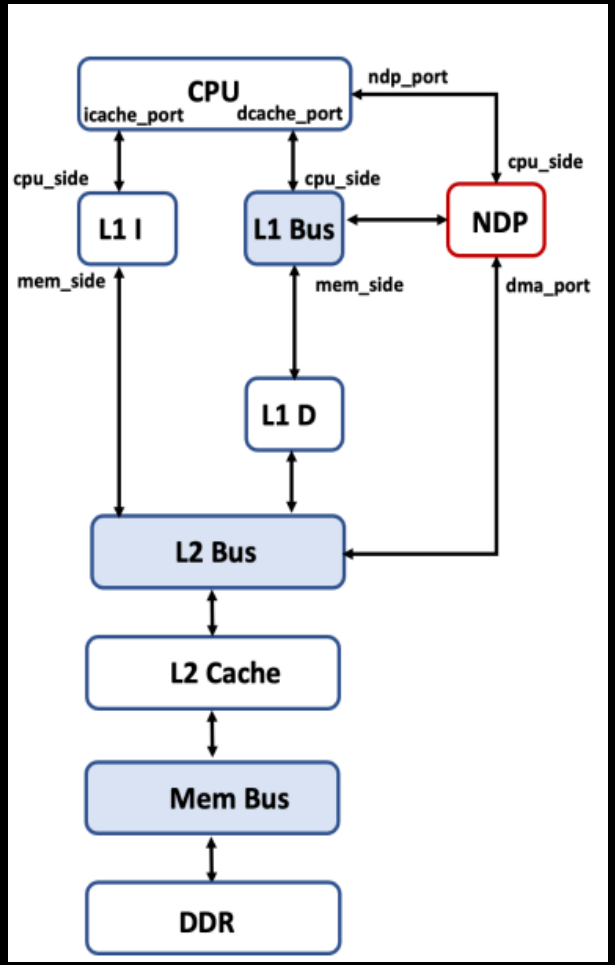
PROPOSAL

- We could aim for near data processing (NDP) to decrease data transfer
- NDP allows for parallelization near data, which gives the benefits of multithreading and prefetching
- All computations are some form of matrix manipulations
  - Streaming data accesses -> No use of caching
  - Ideal candidate for NDPs
- We first explore some related work on NDP

RELATED WORK - NEAR DATA PROCESSING

MEMORY-SIDE ACCELERATION FOR IRREGULAR DATA

- The work of [1] proposes a NDP module MAID that communicates with the L2 and L1 cache, to speedup sparse matrix operations
- Functions like a prefetcher, but it only prefetches data that will be used by the CPU
- It computes the non-zero elements of matrices, fetches and stores them in L1 with a flag to indicate if the data validity



[1] Pranathi Vasireddy, Krishna Kavi, Alex Weaver, and Gayatri Mehta. 2024. Streaming Sparse Data on Architectures with Vector Extensions using Near Data Processing. In Proceedings of the International Symposium on Memory Systems (MEMSYS '23). Association for Computing Machinery, New York, NY, USA, Article 16, 1–12. https://doi.org/10.1145/3631882.3631898

MEMORY-SIDE ACCELERATION FOR IRREGULAR DATA

- The close proximity to L2 can reduce cache pollution at L1 cache since MAID handles metadata
- The sparse data processed by MAID can bypass the intermediate caches and be pushed directly into the L1 cache of the CPU

Table 2: Speedup against In-order baselines on a 512x512 matrix with 10% to 90% sparsity.

	Buffer size: 8			Buffer size: 128		
Vector length	**2**	**4**	**8**	**2**	**4**	**8**
Scalar CPU	0.95x to 0.92x	1.44x to 1.3x	1.92x to 1.56x	1.23x to 1.15x	2.18x to 1.83x	3.53x to 2.39x
SIMD CPU	1.57x to 1.53x	1.75x to 1.68x	1.94x to 1.80x	2.03x to 1.91x	2.65x to 2.36x	3.56x to 2.76x
Dense CPU	1.27x to 10.70x	1.42x to 11.36x	1.56x to 11.26x	1.63x to 13.40x	2.16x to 15.97x	2.89x to 17.23x

Table 3: Speedup against Out-of-order baselines on a 512x512 matrix with 10% to 90% sparsity.

	Buffer size: 8			Buffer size: 128		
Vector length	**2**	**4**	**8**	**2**	**4**	**8**
Scalar CPU	1.73x to 1.72x	2.18x to 2.13x	2.45x to 2.00x	1.95x to 1.15x	2.46x to 2.32x	3.88x to 3.14x
SIMD CPU	1.53x to 1.23x	1.37x to 1.42x	1.28x to 1.37x	1.77x to 1.39x	1.55x to 1.47x	2.02x to 1.88x
Dense CPU	0.98x to 8.56x	0.95x to 8.18x	0.90x to 7.48x	1.13x to 9.69x	1.07x to 8.94x	1.44x to 10.29x

[1] Pranathi Vasireddy, Krishna Kavi, Alex Weaver, and Gayatri Mehta. 2024. Streaming Sparse Data on Architectures with Vector Extensions using Near Data Processing. In Proceedings of the International Symposium on Memory Systems (MEMSYS '23). Association for Computing Machinery, New York, NY, USA, Article 16, 1–12. https://doi.org/10.1145/3631882.3631898

SPARSE-P

- Sparse P [2] performs SpMV execution in Processing In Memory (PIM) cores
- It uses the parallelizability of Matrix Vector (MV) operations to offload execution to these PIM chips to compute partial results

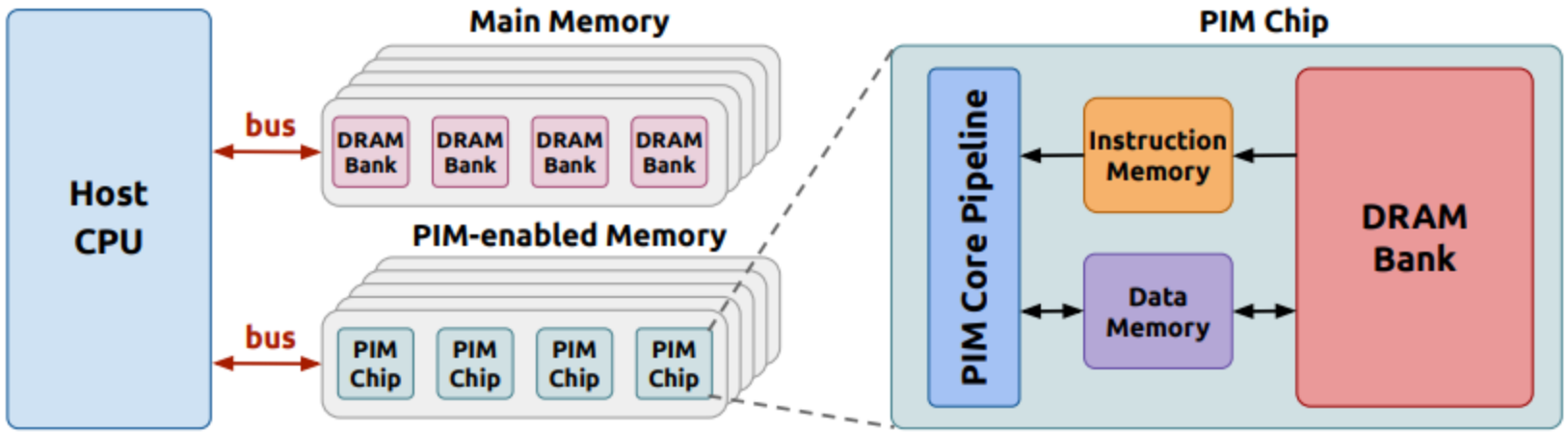


Fig. 3. High-level organization of a near-bank PIM architecture.

[2] Christina Giannoula, Ivan Fernandez, Juan Gómez Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. 2022. SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures. Proc. ACM Meas. Anal. Comput. Syst. 6, 1, Article 21 (March 2022), 49 pages. https://doi.org/10.1145/3508041

SPARSE-P

- Sparse P employs a unique partitioning technique to allow load balancing – ensuring equal usage of all PIM chips
- SparseP typically employs a row-wise or block-wise load balancing technique
- Even if the data is stored in a different PIM chip, it can be accessed transparently by other chips

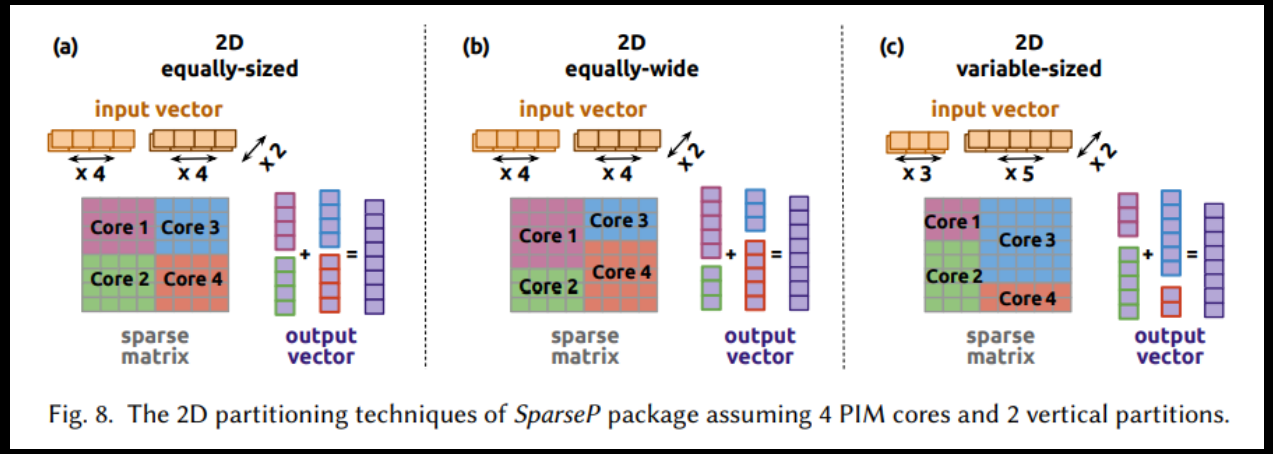


Fig. 8. The 2D partitioning techniques of *SparseP* package assuming 4 PIM cores and 2 vertical partitions.

[2] Christina Giannoula, Ivan Fernandez, Juan Gómez Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. 2022. SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures. Proc. ACM Meas. Anal. Comput. Syst. 6, 1, Article 21 (March 2022), 49 pages. https://doi.org/10.1145/3508041

OBSERVATIONS

- We observe a benefit from offloading computations to the memory for streaming data accesses
- Streaming accesses for large matrices is an ideal candidate for NDPs
- Additionally, a speedup is observed from reducing redundant computations

FUTURE WORK

- Having identified the bottleneck to be the large data movement, we look to minimize this by implementing NDP
- We aim to implement specialized power efficient computational units on the logic layer of 3D stacked memories, for handling matrix operations
- Our goal is to design a domain specific system to accelerate CFDs

THANK YOU