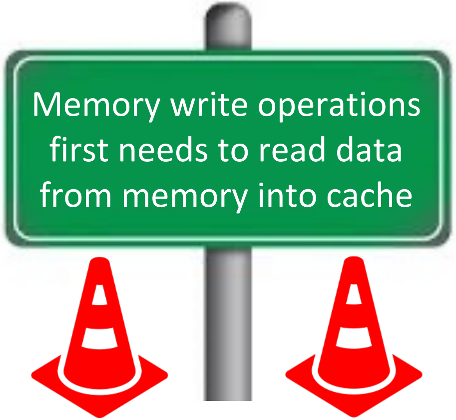
Impact of Write-Allocate Elimination for Graph Analytics on Ookami

Yan Kang (PSU, ybk5166@psu.edu),
Sayan Ghosh (PNNL, sg0@pnnl.gov)
Mahmut Kandemir (PSU)

2nd Ookami User Group Meeting

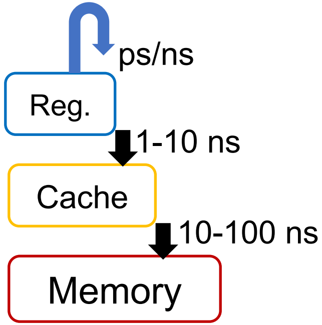
Motivation



Memory write operations first needs to read data from memory into cache

a[...] = b[…]

	Cache	Memory
t=0	☒ Write miss	a[------------]
t=1	☑ Allocate line	a[------------]
t=2	↩ Copy	a[------------]



If the cache line is going to be overwritten anyway, what is the point of the read?

How can we get rid of this spurious read operation and what would it take?

Will it improve performance of applications, by how much?

What is Write-Allocate Elimination?

Write-Allocate: Allocate a cache line for new data

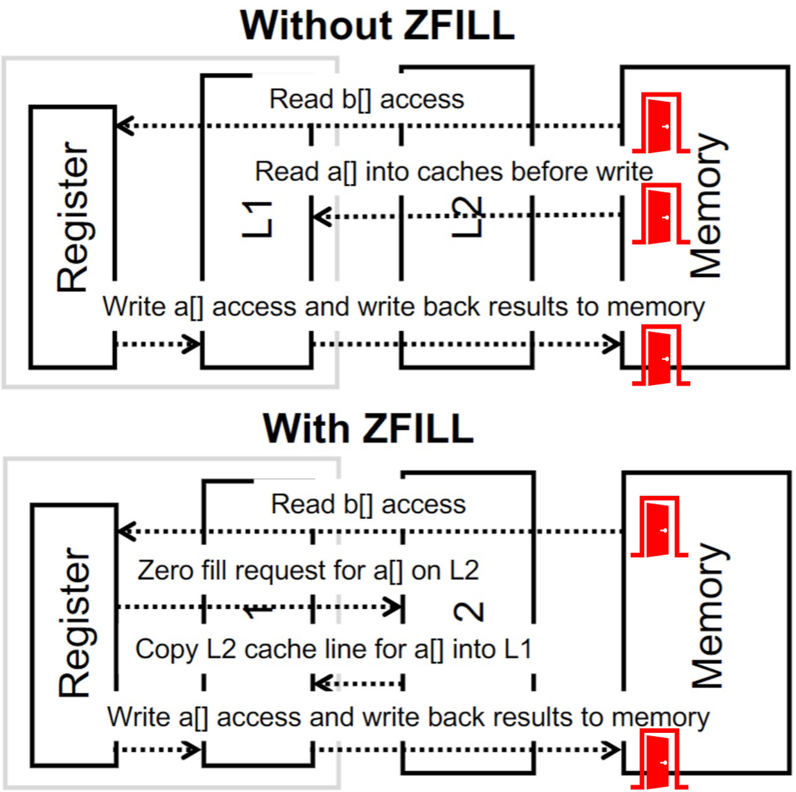
Evasion: [Hardware detects if cache line is going to be overwritten] store cache line directly in memory (Intel, non-temporal stores, compiler hints or automatic SpecI2M)

Elimination: [Hardware detects if cache line is going to be overwritten] directly write an L2 cache line with zeroes, processor loads cache line avoiding memory read

Fujitsu A64FX: Performs "zero filling" through a special 64-bit instruction (DC ZVA) in the ARMv8-A

Read Dr. Georg Hager's blog post and paper:
https://blogs.fau.de/hager/archives/8997
https://onlinelibrary.wiley.com/doi/10.1002/cpe.6512

Zero Filling in Fujitsu A64FX

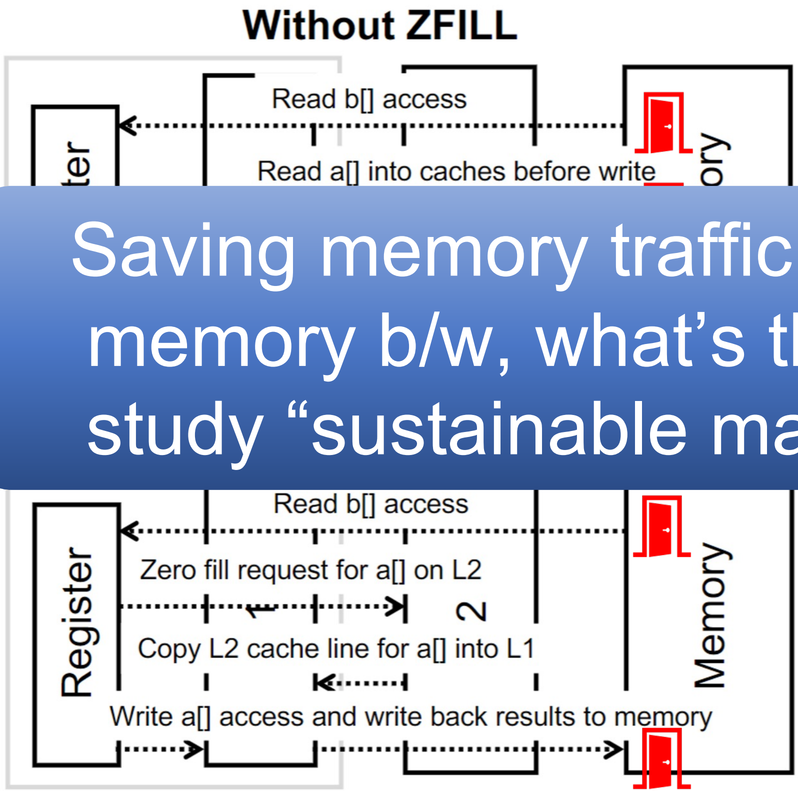


“zero fill” on L2 Cache:
Upon receiving the DC ZVA request, the L2 cache secures the cache line corresponding to the specified virtual address and writes zero data

“zero fill” on L1 Cache:
zero data is written after data in the L1 cache is written back to the L2 cache.

🚪 Memory access

Zero Filling in Fujitsu A64FX



Saving memory traffic means improving memory b/w, what’s that benchmark to study “sustainable main memory b/w”?

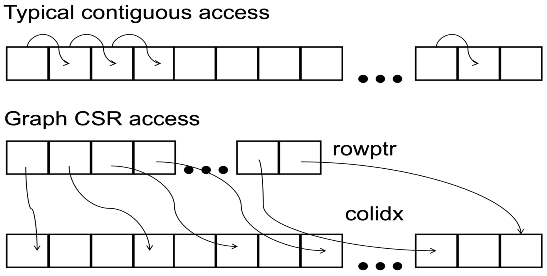
the

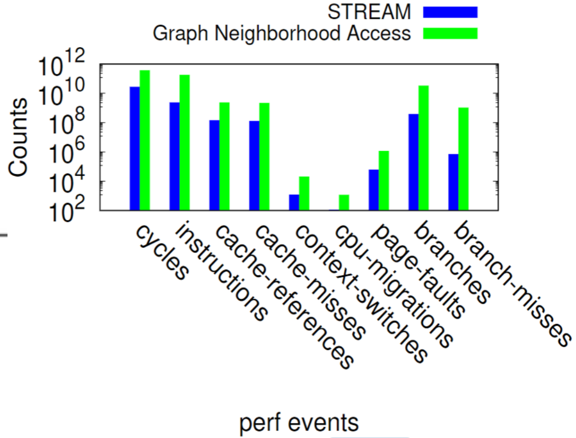
“zero fill” on L1 Cache:
zero data is written after data in the L1 cache is written back to the L2 cache.

Memory access

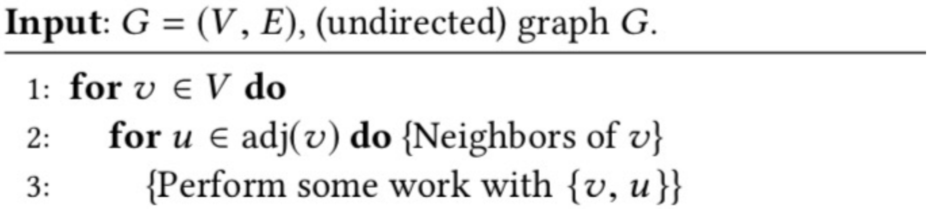
Benchmarking decisions

- STREAM is “best case” memory b/w benchmark
  - Does not represent irregular cases, most applications
- Graphs – irregular memory accesses
  - Applications perform repetitive *neighborhood accesses*
- NEVE is a benchmark, like STREAM for graphs (has COPY, SUM and MAX) - |V|*|E|*#ops / t





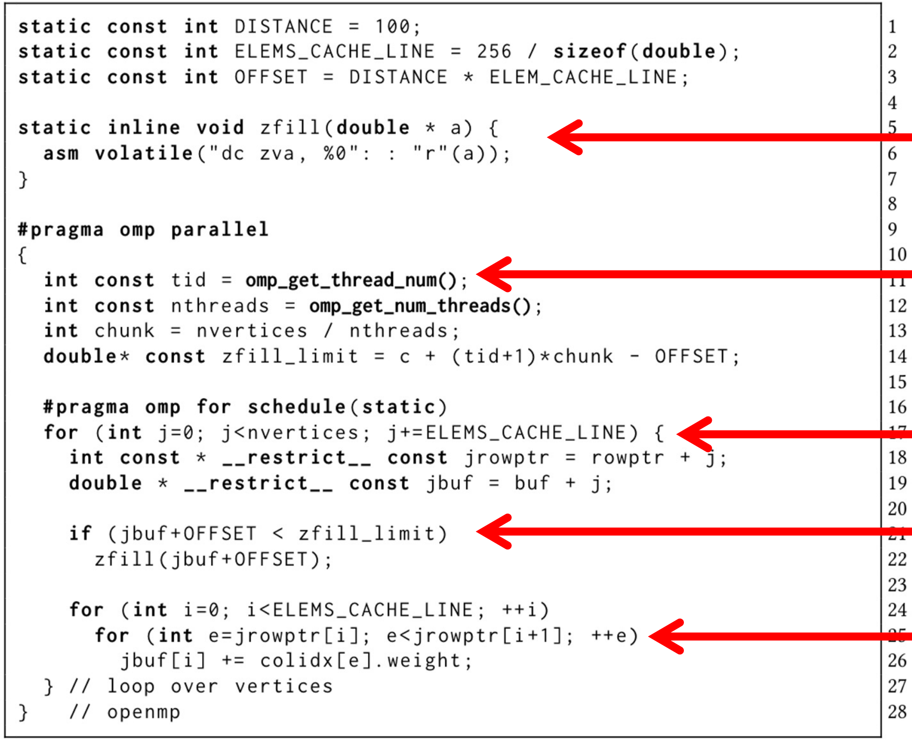
Can return MB/s!



```
Input: $G = (V, E)$, (undirected) graph $G$.
for $v \in V$ do
    for $u \in \mathrm{adj}(v)$ do {Neighbors of $v$}
        {Perform some work with $\{v, u\}$}
```

Sayan Ghosh, Nathan R. Tallent, and Mahantesh Halappanavar. "Characterizing performance of graph neighborhood communication patterns." *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2021): 915-928.

Explicit “Zero Fill” formulation for graph neighborhood accesses



```
static const int DISTANCE = 100;
static const int ELEMS_CACHE_LINE = 256 / sizeof(double);
static const int OFFSET = DISTANCE * ELEM_CACHE_LINE;

static inline void zfill(double * a) {
  asm volatile("dc zva, %0": : "r"(a));
}

#pragma omp parallel
{
  int const tid = omp_get_thread_num();
  int const nthreads = omp_get_num_threads();
  int chunk = nvertices / nthreads;
  double* const zfill_limit = c + (tid+1)*chunk - OFFSET;

  #pragma omp for schedule(static)
  for (int j=0; j<nvertices; j+=ELEMS_CACHE_LINE) {
    int const * __restrict__ const jrowptr = rowptr + j;
    double * __restrict__ const jbuf = buf + j;

    if (jbuf+OFFSET < zfill_limit)
      zfill(jbuf+OFFSET);

    for (int i=0; i<ELEMS_CACHE_LINE; ++i)
      for (int e=jrowptr[i]; e<jrowptr[i+1]; ++e)
        jbuf[i] += colidx[e].weight;
  } // loop over vertices
}   // openmp
```

Listing 1: Graph neighbor access pseudocode leveraging “zero fill” for degree accumulation.

Explicit assembly to invoke DC ZVA

Each thread works on fixed chunk of iterations over |V| (work is variable)

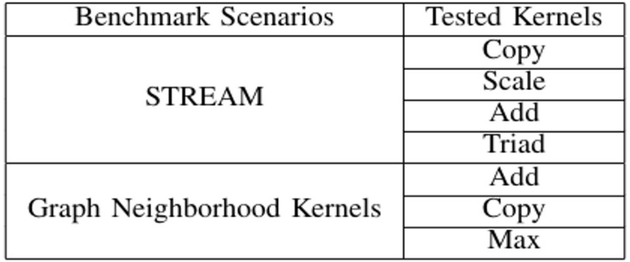
Block outermost loop over vertices

Invoke zero fill in strides larger than L2 prefetch distance

Inner loop, where the zfill virtual address will be invoked several times (trip count unknown)

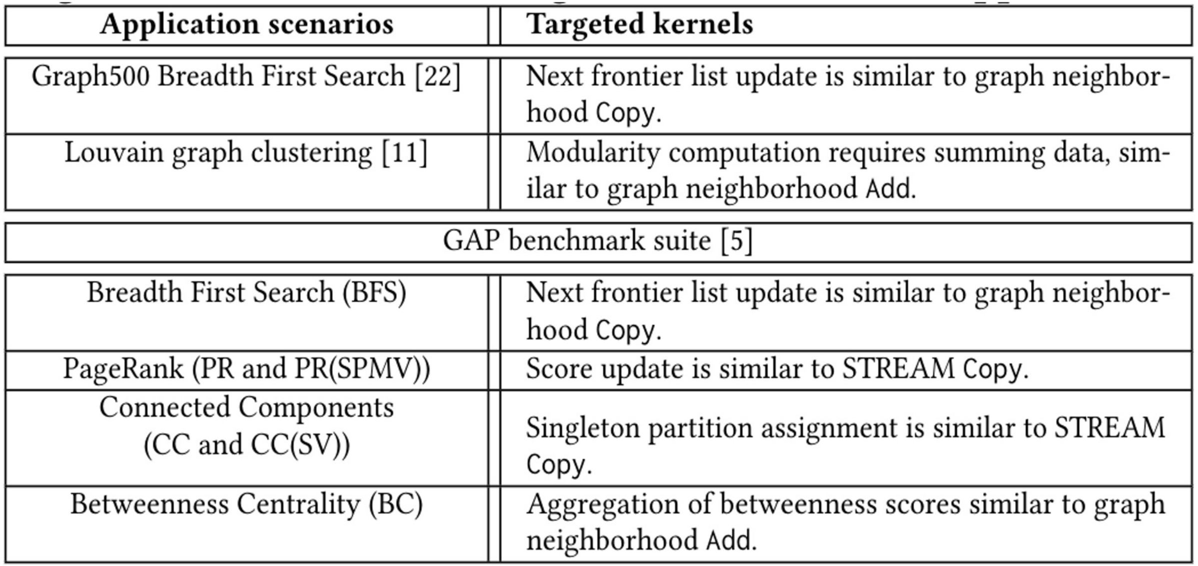
https://gitlab.com/arm-hpc/training/arm-sve-tools/-/tree/master/06_A64FX/02_stream/04_stream_zfill

Benchmarks and applications for evaluations

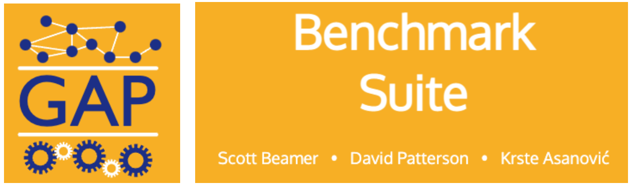


Benchmark Scenarios	Tested Kernels
STREAM	Copy
	Scale
	Add
	Triad
Graph Neighborhood Kernels	Add
	Copy
	Max

Expect STREAM to be the best case!



Application scenarios	Targeted kernels
Graph500 Breadth First Search [22]	Next frontier list update is similar to graph neighborhood Copy.
Louvain graph clustering [11]	Modularity computation requires summing data, similar to graph neighborhood Add.
GAP benchmark suite [5]	
Breadth First Search (BFS)	Next frontier list update is similar to graph neighborhood Copy.
PageRank (PR and PR(SPMV))	Score update is similar to STREAM Copy.
Connected Components (CC and CC(SV))	Singleton partition assignment is similar to STREAM Copy.
Betweenness Centrality (BC)	Aggregation of betweenness scores similar to graph neighborhood Add.



http://gap.cs.berkeley.edu/benchmark.html

+

https://github.com/sg0/louvain-offload
https://github.com/sg0/gapbs
https://github.com/sg0/graph500
https://github.com/sg0/neve

STREAM benchmark evaluations (GCC, ARM and FCC)

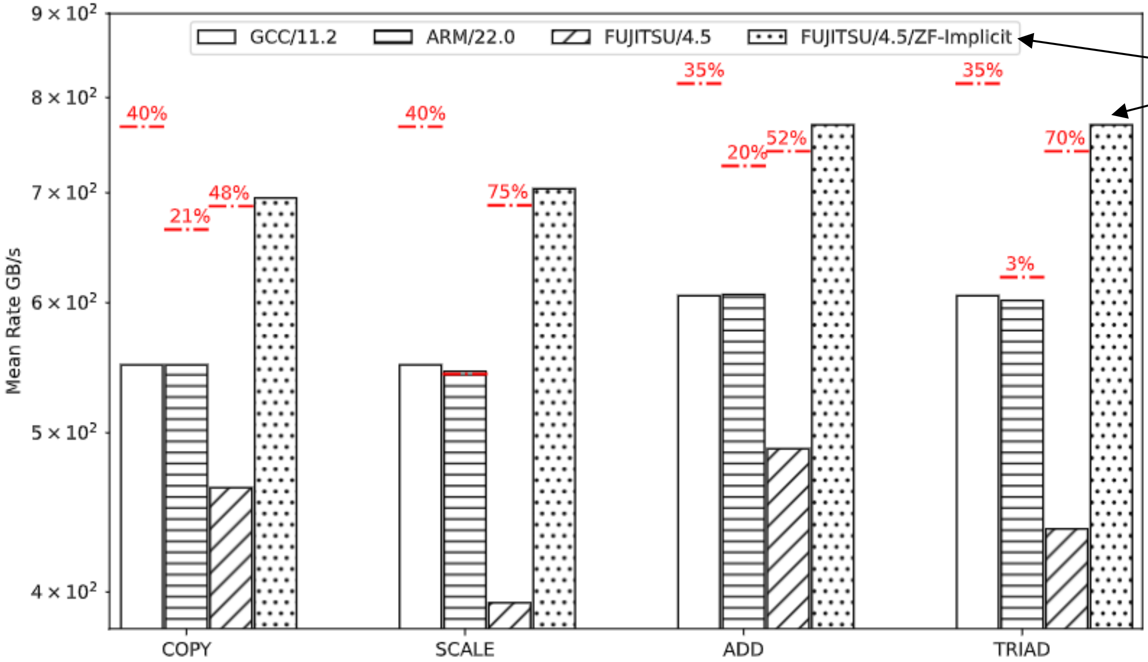


Figure 5: Performance of STREAM (GB/s, *more* is better) across compilers for regular and "zero fill" (red broken lines) versions.

Fujitsu has a compiler option (-Kzfill), referred as implicit version [does not work for C++ compiler]

All compilers demonstrate improvements, FCC up to 70%!

Graph benchmark evaluations (GCC, ARM and FCC)

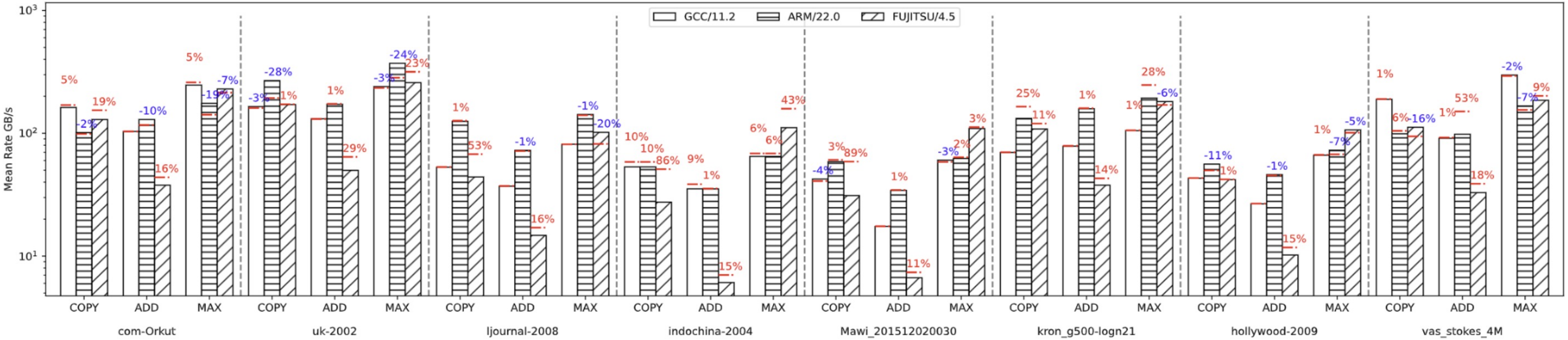


Figure 7: Performance of graph neighborhood kernels (GB/s, *more* is better) across compilers and graphs for regular and "zero fill" versions. Text in blue indicates performance degradation in percentage for "zero fill" version, whereas red indicates a relative performance improvement.

- Used different graphs – implies different structure/work-per-loop
- ZFILL: degradation of up to 28% but also up to 90% improvement (FCC)
- Fujitsu: Irregularities with ADD kernel – >4x memory writes, 3 extra instructions to perform ADD operation compared to GCC/ARM!

Graph Application Evaluations

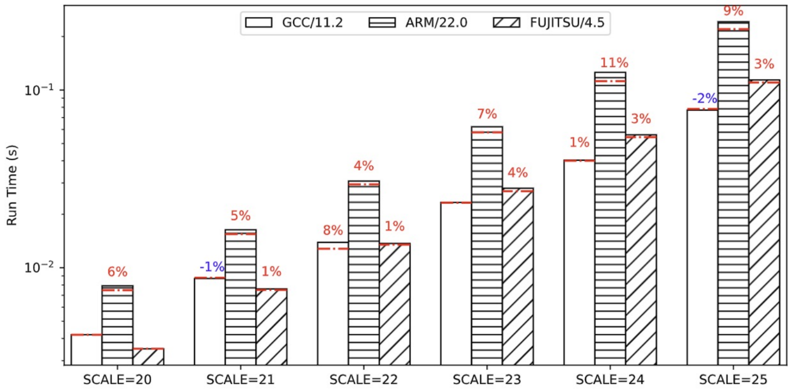


Figure 9: Execution times of Graph500 BFS with % improvements for explicit "zero fill" version against scale 20–25 Kronecker graphs.

- Does not improve performance where there is limited work in the ZFILL section
- ~10% improvement when there is sufficient work

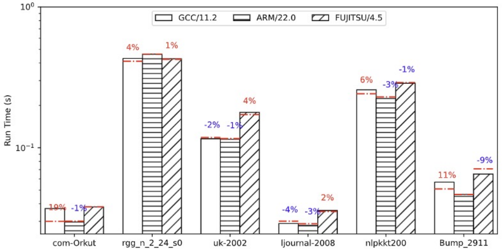


Figure 10: Performance of GAP BFS benchmark for regular and "zero fill" (% improvements) under different graphs.

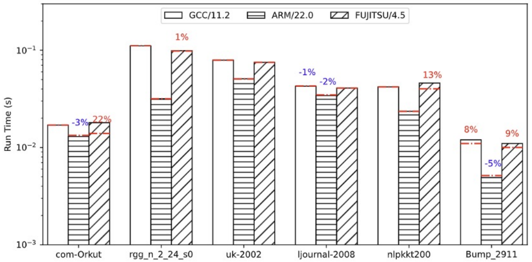


Figure 13: Performance of GAP CC benchmark for regular and "zero fill" (% improvements) under different graphs.

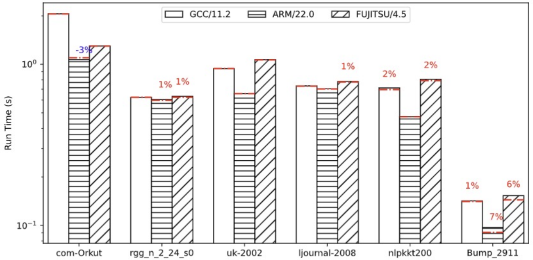


Figure 11: Performance of GAP PR benchmark for regular and "zero fill" (% improvements) under different graphs.

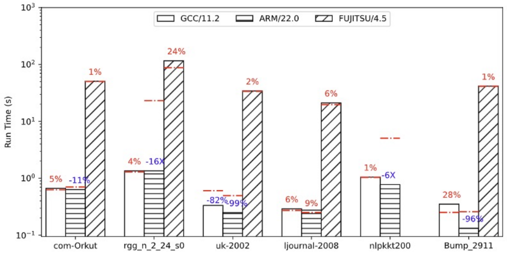


Figure 15: Performance of GAP BC benchmark for regular and "zero fill" (% improvements) under different graphs.

Observations

- NEVE exhibit about 2–5x performance degradation compared to STREAM
- Fujitsu ZFILL-implicit on Graph500 BFS demonstrate 7–17% improvement
  - Compared to 3–11% improvement for explicit version (compiler can win here!)
- Median improvement of 5–9% GAP PR and CC benchmarks

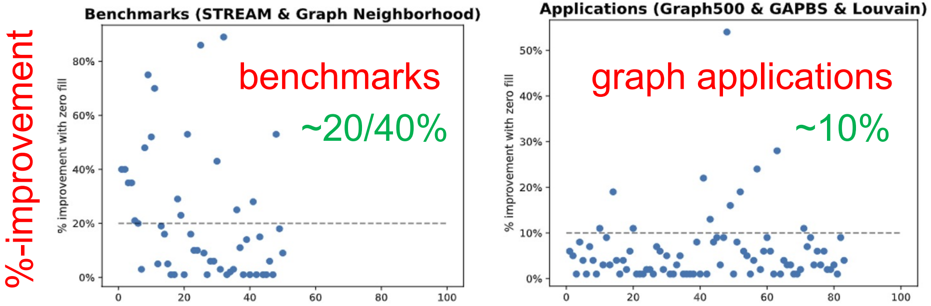
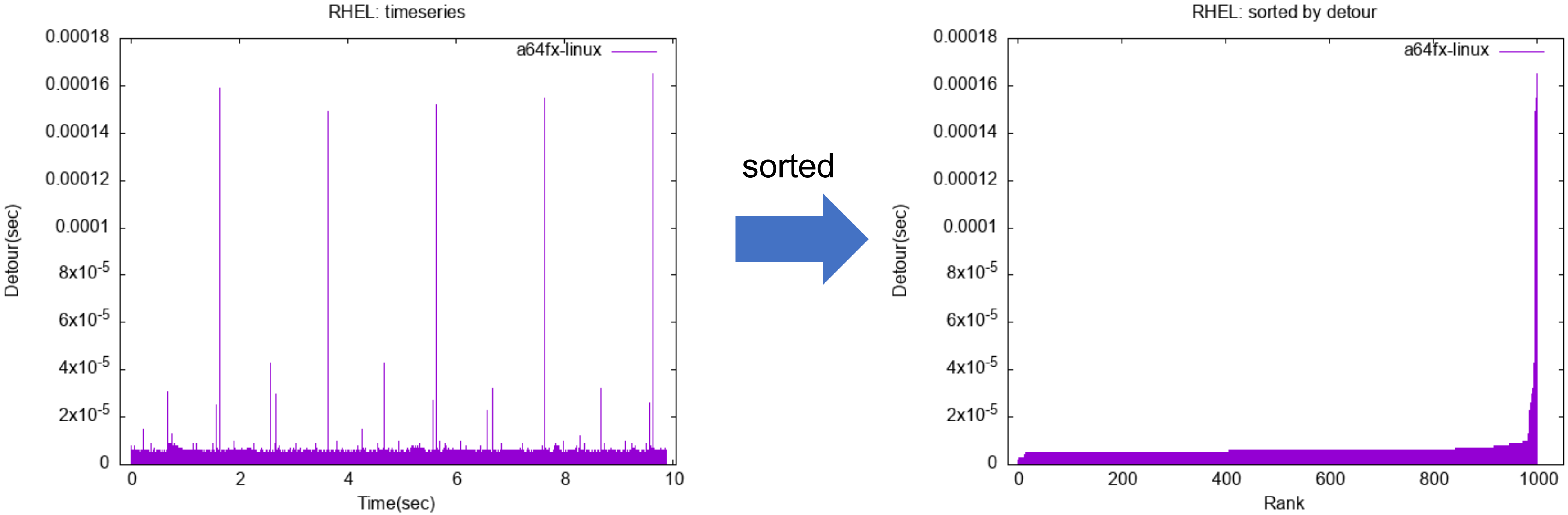


Figure 17: Zero Fill % improvement quantities for benchmarks and applications across various graphs and compilers.

Performance variabilities for irregular workloads on Ookami



sorted

Selfish Detour benchmark indicates noise

FX700 vs. FX1000 (in terms of performance events availability)

L2 events, FX700 (Ookami)

```
> [sayaghosh@fj002 ~]$ perf list | grep -i l2
l2d_cache OR armv8_pmuv3_0/l2d_cache/                  [Kernel
PMU event]
l2d_cache_refill OR armv8_pmuv3_0/l2d_cache_refill/  [Kernel
PMU event]
l2d_cache_wb OR armv8_pmuv3_0/l2d_cache_wb/        [Kernel
PMU event]
l2d_tlb OR armv8_pmuv3_0/l2d_tlb/                      [Kernel
PMU event]
l2d_tlb_refill OR armv8_pmuv3_0/l2d_tlb_refill/        [Kernel
PMU event]
l2i_tlb OR armv8_pmuv3_0/l2i_tlb/                       [Kernel
PMU event]
l2i_tlb_refill OR armv8_pmuv3_0/l2i_tlb_refill/         [Kernel
PMU event]
```

L2 events, FX1000 (Fugaku) [via Jens Domke, RIKEN]

```
[u10016@e29-3210s ~]$ perf list | grep -i l2

ea_l2 [This event counts energy consumption per cycle of L2 cache]
l2_miss_count [This event counts the number of times of L2 cache miss]
l2_miss_wait [This event counts outstanding L2 cache miss requests per cycle]
l2d_cache
l2d_cache_mibmch_prf [an L2 cache refill buffer allocated by demand access]
l2d_cache_refill
l2d_cache_refill_dm [This event counts L2D_CACHE_REFILL caused by demand access]
l2d_cache_refill_hwprf [This event counts L2D_CACHE_REFILL caused by hardware prefetch]
l2d_cache_refill_prf [This event counts L2D_CACHE_REFILL caused by software or hardware
l2d_cache_swap_local [This event counts operations where demand access hits an L2 cache
l2d_cache_wb
l2d_swap_dm [This event counts operations where demand access hits an L2 cache
l2d_tlb
l2d_tlb_refill
l2i_tlb
l2i_tlb_refill
l2hwpf_inj_alloc_pf [This event counts allocation type prefetch injection requests to L2
l2hwpf_inj_noalloc_pf [L2 cache generated by hardware prefetcher]
l2hwpf_other [This event counts prefetch requests to L2 cache generated by the other
l2hwpf_stream_pf [This event counts streaming prefetch requests to L2 cache generated by for L1D cache, L2
cache and memory access
L1D cache, L2 cache and memory access]
ld_comp_wait_l2_miss
ld_comp_wait_l2_miss_ex
l2_pipe_comp_all [This event counts completed requests in L2 cache pipeline]
l2_pipe_comp_pf_l2mib_mch [an L2 cache refill buffer allocated by demand access]
l2_pipe_val [This event counts valid cycles of L2 cache pipeline]
```

Thanks

- PNNL LDRD Data Model-Convergence
- DOE ASCR Advanced Memory to Support Artificial Intelligence for Science (AIAMS, PI: Andrés Márquez, PNNL)
- Penn State HPCL (Prof. Mahmut Kandemir)
- Ookami testbed support (Eva and team)