Seminar in Computer Architecture

PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture

Junwhan Ahn, Sungjoo Yoo, Onur Mutlu† and Kiyoung Choi
Seoul National University †Carnegie Mellon University


Presented by: Georg Streich
Paper

Problem

- Memory accesses are among the largest bottlenecks in both execution speed and energy efficiency for many applications today.
- Some applications can mitigate this bottleneck by exploiting spatial and temporal data locality through the cache hierarchy (e.g. optimized matrix multiplication).
- Other applications, like graph algorithms, don’t exhibit much data locality.
PageRank Algorithm

```cpp
parallel_for (v: graph.vertices) {
    v.pagerank = 1.0 / graph.num_vertices;
    v.nextPagerank = 0.15 / graph.num_vertices;
}

count = 0;

while (++count < maxIteration && diff > e) {
    parallel_for (v: graph.vertices) {
        delta = 0.85 * v.pagerank / v.out_degree;
    }

    for (w: v.successors) {
        atomic w.nextPagerank += delta;
    }

    parallel_for (v: graph.vertices) {
        atomic diff += abs(v.nextPagerank - v.pagerank);
        v.pagerank = v.nextPagerank;
        v.nextPagerank = 0.15 / graph.num_vertices;
    }
}
```

Figure 1: Pseudocode of parallel PageRank computation.

- Only one atomic add operation is applied to fetched value
- Memory accesses don’t have much spatial, temporal locality
- Memory access cannot be amortized
Problem

• We have to bring computation **closer to the memory** in order to make these types of algorithms more efficient

• New manufacturing technologies (3D stacking) enable us to combine memory and logic in a **cost effective** way
  – Enables sensible **processing in memory** (PIM) hardware
  – This technically allows us to do what we want

• How do we **enable** applications to use this technology?
Goals

● Make new hardware manufacturing capabilities accessible to applications

● Enable applications to use PIM

● Make PIM usable through the traditional *sequential programming* model

● Do this with *minor modifications* to existing systems
Key Mechanism

• Add simple PIM enabled instructions (PEIs) to existing instruction sets via ISA extensions
  - These instructions then can be executed on a computation unit close to the memory
• Make PEIs work almost *seamlessly* alongside normal instructions
• PEIs can be used via intrinsics
• A smart compiler could emit PEIs at the right places
• Much like vector instructions
Challenges

- For seamless execution of PEIs promises of memory model need to be kept
  - Atomicity
  - Cache coherency
- PEIs should use the existing virtual memory abstraction
- Execution of PEIs should be scheduled in a sensible way
  - Performance of PEIs should be on par with normal execution for all workloads
  - Doing PIM when we have good data locality can increase memory communication overhead
  - We want to execute PEIs on-chip in that case
Locality-Aware Execution

- PageRank on graphs with increasing size (y-axis)
- PIM decreases performance for small input sizes
- Implementation should be aware of this
- Need locality-aware execution

![Graph showing performance improvement with PIM](image)

**Figure 2:** Performance improvement with an in-memory atomic addition operation used for the PageRank algorithm.
Key Ideas

- Bring computation closer to the data with PIM
- Integrate PIM into the existing programming model via PEIs
- Make PEIs usable for all workloads
Implementation
for (v: graph.vertices) {
    x = weight * v.rank;
    for (w: v.successors) {
        atomic w.next_rank += x;
    }
}
for (v: graph.vertices) {
    x = weight * v.rank;
    for (w: v.successors) {
        __pim_add(&w.next_rank, x);
    }
}

- Depending on our algorithm different instructions could be useful (__pim_min, __pim_and, etc.)
- PEIs can have input, output operands
- PEIs can only operate on one cache-block at a time
Architecture Overview

● Architecture has two main tasks
  – Enable execution of PEIs
  – Schedule the execution of PEIs in sensible way

● Paper implements PIM alongside hybrid memory cube (HMC) technology
  – Adaptable to other memory technologies
3D Stacking Technology

- Stack of separate silicon dies, including memory and logic dies
- Dies are vertically connected by through silicon vias (TSV)
- On-chip communication over TSV much more efficient than between individual chips
- Multiple competing products
  - Hybrid memory cube (HMC) (discontinued)
  - High bandwidth memory (HBM)
  - Newer DDR standards, e.g. DDR5
- Host-memory communication via packet based protocol in case of HMC
Architecture Overview

- Two components (PCU, PMU) are introduced
  - **PIM computation unit** (PCU) responsible for execution of PEIs
  - One PCU for every HMC vault and every host processor core
  - **PIM management unit** (PMU) responsible for scheduling PEIs
  - Single PMU placed near the last level cache
  - PMU is shared between the cores
PEI Execution

- PEIs are issued to host-side PCU by host-processor core
- PCU then talks to PMU to manage the execution of that PEI
- PEIs can be either scheduled on host-side or memory-side PCU
- PEI is executed
- Results are passed back to the host-processor through host-side PCU
- PMU needs to ensure
  - Atomicity
  - Cache coherence
  - Locality-aware execution
Atomicity

- Atomicity only between PEIs, but not between PEIs and normal instructions
  - Assume memory location is accessed many times by PEI operation before and after a normal instruction has to access it
  - Paper introduces pfence instruction to separate PEIs and normal instructions, stops execution until all PEIs have completed

```cpp
parallel_for (v: graph.vertices) {
    v.pagerank = 1.0 / graph.num_vertices;
    v.next_pagerank = 0.15 / graph.num_vertices;
}

count = 0;

do {
    parallel_for (v: graph.vertices) {
        delta = 0.85 * v.pagerank / v.out_degree;
        for (w: v.successors) {
            atomic w.next_pagerank += delta;
        }
    }
    diff = 0.0;
    parallel_for (v: graph.vertices) {
        atomic diff += abs(v.next_pagerank - v.pagerank);
        v.pagerank = v.next_pagerank;
        v.next_pagerank = 0.15 / graph.num_vertices;
    }
} while (++count < max_iteration && diff > e);
```

Figure 1: Pseudocode of parallel PageRank computation.
Atomicity

- Atomicity PEIs executed on the memory-side can be handled by DRAM-controller
  - Controller can schedule all memory instructions for PEI as inseparable group
- Atomicity of host-side PEIs is harder
  - PEIs only access individual cache blocks
  - Could have a reader-writer lock for every cache block
  - Too expensive, so we conservatively approximate this using what the paper calls the PIM directory
PIM Directory

- Located inside the PMU
- Contains a number of hardware based reader-writer locks, each responsible for multiple cache blocks
- PCU requests read, write access for each PEI
- PEI is scheduled when cache block is available
Cache Coherence

- **Host-side** PCUs share the L1-cache with the host processor
  - Don’t need to think about cache coherence for them
- Values accessed by **memory-side** PCUs might be stale, there could be a modified versions of them in cache
  - Before execution PMU requests **invalidation** for writer PEIs and **write back** for reader PEIs
  - This is not too bad, as this only needs to be done in **infrequent** cases, we expect PEIs to be executed on the host if it’s values are in cache
Locality-Aware Execution

- If PEI has high data locality it should be scheduled on the host-side PCU
- Generally cache blocks that are in cache have high locality, we can use this information to our benefit
- Paper introduces locality monitor
  - A lot like tag array of the last level cache
  - Locality monitor mostly mirrors last-level cache-policy, but
    - Also updates entries that get accessed by PEIs multiple times
    - Only store partial tags (xor folded) as we can allow some amount of errors here as this is only a heuristic
Virtual Memory Support

- PEIs issued by host-processor
  - PEIs use virtual addresses
  - Addresses can be translated by TLB like normal
- PCUs operate only on physical addresses
  - No need to translate addresses on the memory side
Results
Results

• Evaluation on different input sizes

• Locality-Aware execution is always on par normal execution (Host-Only)

• As expected PIM improves performance mainly for large input sizes

Figure 6: Speedup comparison under different input sizes.
Takeaways

- Integration of PIM into the *sequential programming* model
- *Locality-aware* execution of PIM operations
- Does this with *small modifications* to existing systems
Questions?
Strengths

- **Minimal support** implementation of PIM that fits into the *existing programming model*
- Using it would only require *small modifications* to existing applications or the use of a PEI enabled compiler
- Works with inputs of all sizes. In contrast to e.g. GPU programming where programmers or a runtime need to decide if the *cost* of offloading computation to the GPU is worth it
- **Pragmatic solution**
  - Avoids a lot of *complexity* by restricting PEIs to single cache blocks while still providing massive improvements for a lot of applications
  - Doesn’t try to implement locality monitoring, read-write access control exactly but makes a good *trade off* between allowing false-positives and an efficient implementation
- **Adaptable** to other memory technologies (e.g. DDR, HBM, PCM)
Weaknesses

• Architecture does not support general purpose computation in PCUs
• PEIs can operate only on single cache blocks
  – Cannot request values that live in different cache blocks
  – Prevents some applications from effectively using this
• Requires modifications to the host-processor
• Memory and processor need to integrate with each other for this to work
• Requires the programmer or compiler to insert PEIs
Related Works
CAIRO

- Proposes a compile-time technique for detecting instructions for which it is **viable** and **beneficial** to be offloaded to in-memory computation
- Instruction selection process
  - Detects instructions, groups of instructions that are eligible to be offloaded
    - Existing atomic read-modify-write (RMW) operations
    - Chains of load-compute-store instructions
  - Approximates **speedup**, **cost** of offloading instructions
    - Measures some **machine-dependent** constant factors using a micro benchmark
    - Then evaluates instructions using that
- Both for **CPU** and **GPU** hosts
GraphPIM


- PIM for graph computing workloads
- Identifies two major performance bottlenecks in current graph algorithm implementations
  - Irregular memory accesses
  - Use of atomic operations
    - PIM can help with both
- Automatically uses PIM for suitable atomic instructions
  - No work needs to be done by programmer or compiler
Opportunistic Computing in GPU Architectures


- Memory hierarchy of GPUs is different from that of traditional CPU, DRAM based systems
  - Multiple last-level caches
  - Data can be on a number of GPU nodes

- Opportunistic near data computation (NDC) for GPUs
  - Not only wants to optimize memory-chip, but also, on-chip communication overhead
  - Can offload computation to last-level caches and also different GPU nodes

- Also identifies load-compute-store instruction chains

- Deals with instructions that require data from multiple different last-level caches
Discussion
Would it make sense to combine this approach with processing using memory (PUM) ideas like AMBIT?

Do they face the same challenges?
How does this compare with the hardware that was recently released by Samsung?

In which ways is this different, similar?
How big is the limitation of PEIs only being able to access single cache blocks?

Are there applications that would benefit from accessing multiple cache blocks?
What other applications might benefit from this?

What applications could benefit from PIM in general?
Seminar in Computer Architecture

PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture

Junwahn Ahn, Sungjoo Yoo, Onur Mutlu† and Kyoung Choi
Seoul National University †Carnegie Mellon University


Presented by: Georg Streich