Improving Hardware Utilization in MAPLE

LEO HAN and YIZHOU GAO, University of Toronto

Recent development in machine learning and graph processing has created a huge demand for processors that can perform sparse matrix operations efficiently. Unlike dense matrix operations which access data contiguously, sparse matrix operations require Indirect Memory Accesses (IMAs) which can have long memory latency.

Various existing techniques aim to mitigate the latency of IMAs through hardware or software design like pre-fetching or Decoupled Access and Execute (DAE). MAPLE [Orenes-Vera et al. 2022] is a NOC-integrated hardware unit that provides DAE support while requiring no modifications to processor and memory architecture. It allows a processor core to offload memory operation asynchronously to a MAPLE instance, thus allowing the processor to continue running while the memory access is in progress. MAPLE maintains loaded data in a FIFO queue for later consumption by an execute core.

However, due to imbalance of workload in most software applications, either the access or the execute thread will run ahead of the other and fill or empty the FIFO queue causing stalls to occur when an access core issues a load to MAPLE or an execute core issues a read to MAPLE. To address this inefficiency in hardware utilization, in this project, we explore allowing cores to switch between access and execute roles as a form of load balancing.

Although due to software overhead, the role-switching implementations show no consistent speed up over the baseline (single access and execute thread), they significantly reduce stalled cycles.

CCS Concepts: • **Computer systems organization → Multicore architectures**.

Additional Key Words and Phrases: multicore, parallel architecture, Decoupled Access and Execute

ACM Reference Format:
Leo Han and Yizhou Gao. 2023. Improving Hardware Utilization in MAPLE. 1, 1 (April 2023), 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Traditionally, applications will stall or block when waiting for long latency memory accesses. To reduce the impact of memory access latency, pre-fetching allows the processor to issue memory request ahead of when the data is needed [Baer and Chen 1991]. By the time processor requires the data, the data is already in cache memory. However, pre-fetching is effective only when future data accesses are predicted correctly. If predicted incorrectly, it can even pollute the cache by ejecting useful data prematurely.

In most applications, when the data access is regular and predictable, i.e. accessing contiguously or in regular strides, pre-fetching works as expected. However, with the rising popularity of sparse linear algebra in machine learning and graph analytic applications, pre-fetching is less effective due to Indirect Memory Accesses (IMAs). A sparse matrix is not stored in full in memory. Instead it only stores the non-zero entries in a contiguous block and have an index array indicating at which column there is a non-zero value row-by-row. One more array is used to indicate where each row starts in the index array. This is known as the compressed sparse row (CSR) format. When performing operations such as sparse matrix multiplication, the access pattern is irregular and pre-fetching will perform poorly.

Instead of predicting future data accesses, Decoupled Access Execute (DAE) [Smith 1982] splits the data access and data consumption of the original program into two different parallel threads at compile time. The access thread will run ahead of the execute thread acting as a perfect pre-fetcher. Both threads are running in program order to ensure data are accessed and consumed in the same order. Previously, to achieve decoupling, either modification to core architecture or cache system [Nguyen and Sanchez 2020] or additional ISA instructions [Crago and Patel 2011] are needed. These modifications makes DAE less attractive to adopt in real hardware.

To circumvent the above mentioned problem, [Orenes-Vera et al. 2022] proposed Memory Access Parallel-Load Engine (MAPLE) which is hardware unit that can be used to provide DAE access with no modifications to processor cores and the ISA. It is a hardware managed FIFO queue connected to processor cores through a NOC. The access thread will issue load request through MAPLE and continue on issuing new requests while MAPLE loads the data from memory. The execute thread will consume the data from MAPLE when it is available. Just like any DAE system, the access thread needs to run ahead of execute thread and both threads need to execute in program order.

As a result of workload imbalances in most software applications as well as having a finite queue size, the MAPLE queue will become full or empty and cause either the access or execute thread to stall. If the task is more access-loaded, the execute thread will eventually run ahead and stall when the queue is empty. If the task is more execute-loaded, the access thread will fill the queue and eventually stall as well.

To improve the hardware utilization of MAPLE in a multi-core setup, we explore different adaptive role-switching schemes. The main idea is to allow role-switching for the threads based on the fullness of the queue such that it avoids consuming when the FIFO is empty and producing when the FIFO is full.

2 ARCHITECTURE

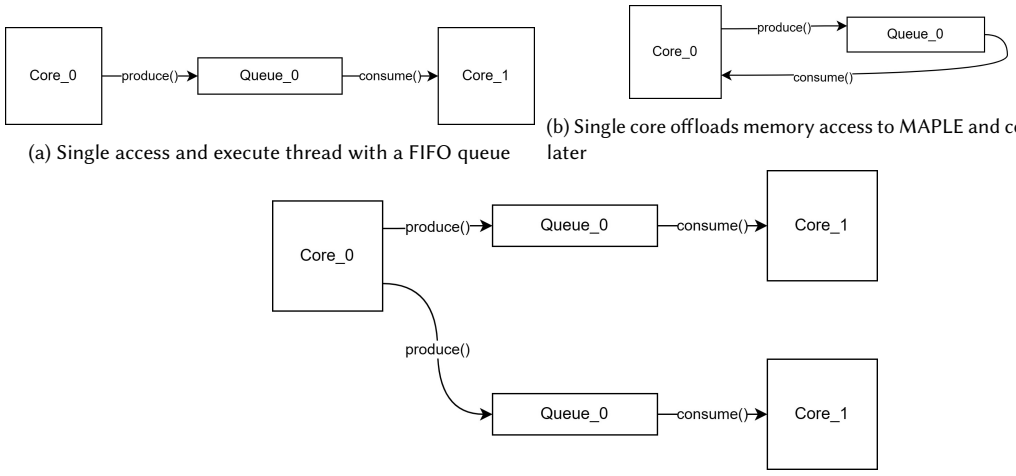
In this section a brief review of MAPLE design is provided. In particular, details regarding how the FIFO queue is implemented, what software API is provided, and how to use MAPLE are covered.

2.1 Hardware Managed FIFO Queue

Each MAPLE instance contains a local scratchpad memory [Orenes-Vera et al. 2022]. It implements the circular FIFO on the scratchpad

Authors' address: Leo Han, leo.han@mail.utoronto.ca; Yizhou Gao, yizhou.gao@mail.utoronto.ca, University of Toronto.

XXXX-XXXX/2023/4-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn



(a) Single access and execute thread with a FIFO queue

(b) Single core offloads memory access to MAPLE and consumes later

(c) Single access thread supplies data to two execute threads with two FIFO queues

Fig. 1. Different way of using MAPLE FIFO queues

memory. The queue size is configurable by the program's need. Each MAPLE instance can support multiple queues at the same time. It is program's choice to decide how many queues needed and how big the queues are. This property allows one MAPLE instance to simultaneously supply data to multiple cores using different FIFO queues. When a queue is full, new access request will be buffered until space is freed up. However, this doesn't block other queue from responding to access request. Same for the case when the queue is empty for read request from execute threads.

A thread can access any queue in a MAPLE instance using Memory-Mapped IO (MMIO). This avoids requiring specialized ISA instructions for supporting the memory operations.

2.2 Software API

MAPLE provides the following software APIs to facilitate memory operations [Orenes-Vera et al. 2022].

Init(id) creates the FIFO queue with id.

Open(id) setups communication with the targeted queue.

Close(id) terminates the targeted queue.

Produce(id, data) allows access thread to push data into FIFO queue.

ProducePtr(id, ptr) allows access thread to request data at particular address. This is the API to offload memory access to MAPLE.

Consume(id) allows execute thread to pop data from queue.

When compiling the decoupled programs with MAPLE support, the above API calls are added to replace normal IMAs.

2.3 How to use MAPLE

To use MAPLE, program can either create decoupled threads and run on different cores (note as **dcpn** see Figure 1a) or have a single thread to use MAPLE to pre-fetch data for its own consumption at a later stage (see Figure 1b).

Due to the support of multiple queue per program, MAPLE allows asymmetric decoupling meaning the number of access threads does not need to match number of execute threads (see Figure 1c). As long as the data are produced and consumed in program order for each queue, the execution will be successful.

This can be useful for application that has high thread-level parallelism to avoid always adding access and execute threads in pair when scaling.

3 METHODOLOGY

As mentioned previously, work load imbalance in access and execute will result in one thread running ahead far enough such that the queue will be empty or full. This will cause the other thread to stall constantly once empty or full (starvation).

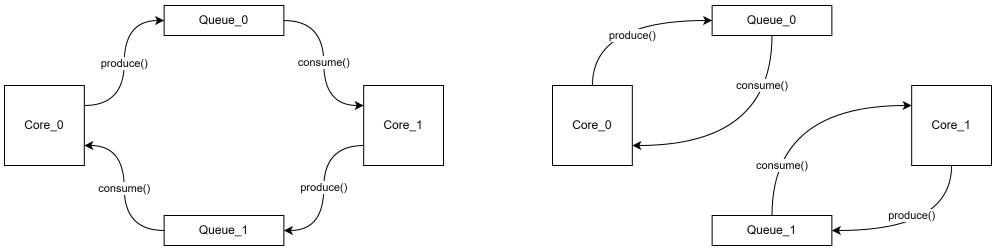
In order to mitigate the under-utilization of less loaded thread, we propose to introduce adaptive role switching schemes to allow access or execute threads to take on a different role to help out instead of waiting.

3.1 Symmetric Switching

The simplest idea is to switch roles when the FIFO is empty or full. If FIFO queue is empty, the execute thread will switch role to become an access thread. If FIFO queue is full, the access thread will switch role to become an execute thread.

However, this simple way of switching can be problematic. For example When FIFO becomes empty, the execute thread is now an access thread. When the FIFO is full with both threads accessing, both threads will switch to being execute thread. As a result, we will result in both threads switching together all the time.

As we know, DAE requires program order execution, with both threads on the same roles, implies constant contention between the two threads on matrices indices as well as head and tail of the FIFO queue. Such overhead on synchronization can be a huge price to pay.



(a) Symmetric switching with alternate queue

(b) Symmetric switching with individual queue

Fig. 2. Different way of using FIFO queues

3.2 Asymmetric Switching

One way to mitigate such contention is to reduce overlap between the two threads on the same role. Instead of having the same switching rule, different threads can maintain its identity as access or execute threads and behave differently.

Each thread is assigned a primary role - primary access and primary execute. For primary access thread, it will switch to execute when FIFO is full but it will switch back to being access thread after FIFO is half way empty. For primary execute thread, it will switch to access when FIFO is empty but it will switch back to being execute when FIFO is half way full.

In this asymmetry switching example, the overlap is 50%. The optimal threshold to trigger switching can vary depending on workload.

However, this approach failed when multiple location requires synchronization. This is because to ensure correct execution order across multiple locations, the thread has to lock a large section of code as critical section thus lowering the parallelism.

3.3 Data Splitting

To avoid contention, the best approach is to exploiting data parallelism and split the data into two different subsets and two corresponding queues (smaller) and each thread manages the access for one and execute for the other (note as **dcpsw** see Figure 2a) or each thread mange the access and execute for each subset (note as **dcpsp** see Figure 2b).

This is also equivalent to having two different programs and the cores multiplexing between the two programs independently. As a result, the asymmetric switching is not necessary.

However, data parallelism is not always present, thus not all problem can adopt this method.

3.4 Implementation

There are four use cases provided by [Orenes-Vera et al. 2022], BFS (breath-first search), EWSD (elementwise sparse product), SPMV (sparse matrix vector multiplication), and SPMM (sparse matrix multiplication), they are all applications with a lot of IMAs.

Because all of the use cases have multiple locations that needs to be synchronized (row index and column index, etc), it is not beneficial to implement the asymmetric switching by itself.

Since all of the use case has data parallelism, we split the data by alternating rows to avoid contention among the two threads. The only overhead added will be polling the status of the queue to determine the timing of switching. As mentioned earlier, with data splitting, asymmetric switching is not necessary, so basic symmetric switching is used.

Effectively we are testing to see if **using smaller but more queues without adding access or execute thread will improve core utilization with the switching scheme introduced.**

4 RESULTS

For each of the use case we tested three different data size, tiny, small, and big. Tiny matrices come with less than 10 rows. Small matrices have about 400 rows. Big matrices contains more than 600 rows.

For fair comparison, all methods, dcpn, dcpsp, and dcpsw use single access and execute thread with a total FIFO capacity of 32 (two-queue configuration will have 16 for each).

For evaluation, we uses the Verilator[1] simulator. It provides cycle-accurate simulation given the RTL model. Both access and execute threads are fired simultaneously and the longer execution time (simulation cycles) of the two is recorded.

The main benefit of the switching is to mitigate stalls. From Table 1 one can see that the read and store stalls due to FIFO full or empty is greatly reduced with the switching implementation. For the big dataset, especially for SPMM, we can see there are a large amount of stalls got reduced. However, it is unclear why the switching is not offering much benefit in the case of BFS with large data set. Further investigation is required.

In Figure 3, the speed up over dcpn on each use case with different data size is presented (with dcpn normalized as 1). For BFS, EWSD, and SPMV, no speed up is achieved over the baseline. Instead, in each of the three test cases, we observe a slow-down which is due to the added overhead for queue state tracking and synchronization across parallel cores. However, for SPMM, 1.3x to 1.6x speed up is observed. BFS, EWSD, and SPMV dcpn versions have relatively fewer FIFO-stalled cycles as a proportion of total cycles compared to SPMM, meaning the benefit of role switching is reduced for BFS, EWSD, and SPMV. For the tests in which a slow down is observed, the reduction in FIFO-stalled cycles does not offset the additional software overhead required for facilitating the switching.

Overall we believe that the switching design can potentially improve the core utilization without increasing the count of cores.

[1] https://www.veripool.org/verilator/

Table 1. Stall Comparison (total read stall + total write stall)

Use case		SPMV			EWSD			SPMM			BFS		
	Size	tiny	small	big	tiny	small	big	tiny	small	big	tiny	small	big
Read	dcpn	9	819	1686	21	2159	7671	214	971	2249	5241	543143	18692105
	dcpsp	8	3552	5337	20	2087	7551	115	34204	57406	559	60447	20270908
	dcpsw	8	792	1580	13	2119	7551	115	779	1779	539	60445	20240763
Store	dcpn	10	22198	48536	22	54966	191988	200	834511	2205759	588	60478	1765720
	dcpsp	10	815	2719	22	2127	7671	119	180229	1908	587	60477	2708279
	dcpsw	10	794	1582	27	2159	7671	199	18019	1888	567	60474	2692876

However, the significant software overhead needs to be addressed for this to be a viable option to adopt.

5 RELATED WORK

To hide the latency in memory accesses, DAE (Decoupled Access Execute) [Smith 1982] separates data access and computation into two different threads at compilation stage. The two threads communicate through a FIFO queue. Whenever there is a data read, the access thread will run ahead of the execute thread and enqueue the data from main memory into the AEQ (Access-to-Execute Queue). The execute thread will then consume the data with minimal memory latency. Whenever the execute thread issues a data store, it will write to the EAQ (Execute-to-Access Queue) and the access thread will then write to the main memory from the queue.

OUTRIDER [Crago and Patel 2011] as one of the early example of DAE extracts *strands*, memory-accesssing and memory-consuming instruction steams, to reduce memory latency. It partition data buffer into multiple queues for communication between different pairs of instruction strand. To ensure synchronization, it added **memProceed** and **memWait** instructions. In contrary, MAPLE does not require additional instruction through dedicated hardware managed queue.

However, there are a few limitations in the naive DAE design: the in-order communication and loss of decoupling due to waiting on output from execute threads. This can be the following cases. First, access thread may need to load the data output by execute thread. But since it can run far ahead, the data may not be updated by execute thread by the time of access. Second case is that there are branch logic dependent on execute thread's output. Last case is that the address to access is dependent on execute threads calculation. In all of these cases, access thread will stall waiting on execute thread to catch up thus losing the benefit of decoupling.

DeSC [Ham et al. 2015] improves upon the basic DAE concept by allowing out-of-order communication, thereby reducing the loss of decoupling. Unlike other speculative methods with complex recovery mechanism for mis-speculation, DeSC uses communication queue for storing load request and a communication buffer to store value. Each load request and stored value is accompanied by program-order id which is used by the **consume** API to locate the correct value to use.

Instead of running a single thread per core either for data access or execution. Mercury [Ham et al. 2019] proposed to have one access core multithreading to supply data to multiple execute cores. It achieves better core and accelerator utilization.

Pipette [Nguyen and Sanchez 2020] took a similar approach by multiplexing between multiple access or execution stages on the same core to achieve better load balance among cores. Which is very similar to our single-threaded approach. In a multicore system, it facilitates both intra-core queue and inter-core queue using existing registers.

However, the above-mentioned DAE techniques either require specialized compute cores or specialized ISA implementations, limiting their applicability on existing hardwares.

To mitigate memory latency, besides DAE approaches, there are some other works worth mentioning.

In OoO processor, it allows out-of-order execution of instruction with an associative instruction window. Instruction that depends on unfinished memory access is left in the window and other instruction are executed. Combined with pre-fetching, this can be very effective. However, OoO processor usually come with more complex circuit and thus less favor of power and area efficiency.

Instead of relying on large instruction window in OoO processor, [Mutlu et al. 2003] proposed to use helper thread to conduct prefetching. This is similar to DAE idea but not as efficient as DAE approaches (with dedicated access thread) for application with large amount of data accessing.

Instead of changing the hardware design, [Koukos et al. 2016] work on the compiler to apply code transformations to allow more efficient memory access.

[Ottoni et al. 2005] proposes an automatic thread extraction method called Decoupled Software Pipelining (DSWP) for creating lasting and concurrent threads to achieve low memory latency. Being a pure software-based approach, it avoids complex hardware design and need for core-to-core communication.

6 CONCLUSION

In this paper, we proposed to introduce role switching to reduce stalling the processor due to FIFO being full or empty when using MAPLE for decoupling memory access and execution. Typically to cope with workload imbalance, one would introduce more threads on the side that is more heavily loaded. However, this would require adding additional cores. We achieved load balancing without adding additional cores by allowing role switching in runtime for access and execute threads by introduce two queues for accessing and

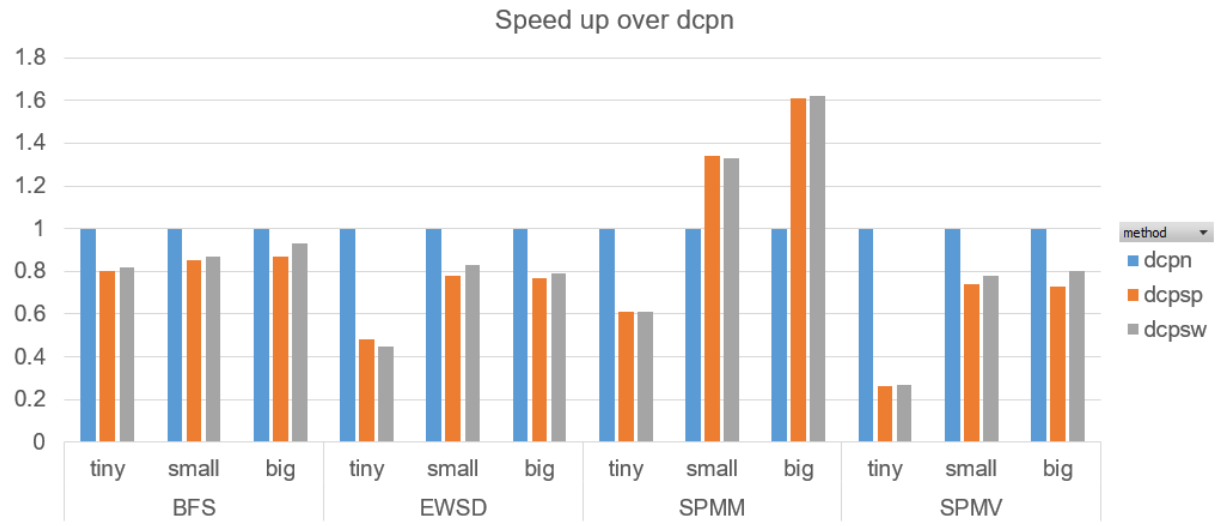


Fig. 3. Speed up over normal decoupling

consuming odd and even rows of the data taking advantage of the configurability in MAPLE. Although there are significant slow down due to software overhead for synchronization, promising result on SPMM indicating that there are benefits that come with the switching implementation.

For future work, such overhead can be reduced by hardware implementation. For example, hardware interrupts can be triggered when FIFO is full or is empty. OS can register the interrupts signal and context switch the access and execute threads for each half of the data. This way, the synchronization overhead is offloaded to hardware and significant speed up over baseline can be possible.

REFERENCES

Jean-Loup Baer and Tien-Fu Chen. 1991. An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing* (Albuquerque, New Mexico, USA) *(Supercomputing '91)*. Association for Computing Machinery, New York, NY, USA, 176–186. https://doi.org/10.1145/125826.125932

Neal Clayton Crago and Sanjay Jeram Patel. 2011. OUTRIDER: Efficient Memory Latency Tolerance with Decoupled Strands. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (San Jose, California, USA) *(ISCA '11)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/2000064.2000079

Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2019. Efficient Data Supply for Parallel Heterogeneous Architectures. *ACM Trans. Archit. Code Optim.* 16, 2, Article 9 (apr 2019), 23 pages. https://doi.org/10.1145/3310332

Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2015. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 191–203. https://doi.org/10.1145/2830772.2830800

Konstantinos Koukos, Per Ekemark, Georgios Zacharopoulos, Vasileios Spiliopoulos, Stefanos Kaxiras, and Alexandra Jimborean. 2016. Multiversioned Decoupled Access-Execute: The Key to Energy-Efficient Compilation of General-Purpose Programs. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 121–131. https://doi.org/10.1145/2892208.2892209

O. Mutlu, J. Stark, C. Wilkerson, and Y.N. Patt. 2003. Runahead execution: an alternative to very large instruction windows for out-of-order processors. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* 129–140. https://doi.org/10.1109/HPCA.2003.1183532

Quan M. Nguyen and Daniel Sanchez. 2020. Pipette: Improving Core Utilization on Irregular Applications through Intra-Core Pipeline Parallelism. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 596–608. https://doi.org/10.1109/MICRO50266.2020.00056

Marcelo Orenes-Vera, Aninda Manocha, Jonathan Balkind, Fei Gao, Juan L. Aragón, David Wentzlaff, and Margaret Martonosi. 2022. Tiny but Mighty: Designing and Realizing Scalable Latency Tolerance for Manycore SoCs *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 817–830. https://doi.org/10.1145/3470496.3527400

Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. 2005. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture* (Barcelona, Spain) *(MICRO 38)*. IEEE Computer Society, USA, 105–118. https://doi.org/10.1109/MICRO.2005.13

James E. Smith. 1982. Decoupled Access/Execute Computer Architectures. *SIGARCH Comput. Archit. News* 10, 3 (apr 1982), 112–119. https://doi.org/10.1145/1067649.801719