Queue Streaming Model, Part 1: Theory

Anup Zope, *Student Member, IEEE,* and Edward Luke, *Senior Member, IEEE,*

Abstract—A model of computation for shared memory parallelism is presented. To address fundamental constraints of modern memory systems, the presented model constrains how parallelism interacts with memory access patterns and in doing so provides a method for design and analysis of algorithms that estimates reliable execution time based on a few architectural parameters. This model is presented as an alternative to modern thread based models that focus on computational concurrency but rely on reactive hardware policies to hide and amortize memory latencies. Since the hardware uses reactive mechanisms and heuristics to deduce the data access requirement of computations, the memory access costs of these threaded programs may be difficult to reliably predict. In this paper, we present the queue streaming model that aims to address these shortcomings by providing a prescriptive mechanism to achieve latency-amortized and predictable-cost data access.

Index Terms—bridging model of computation, queue streaming, performance predictability and portability

1 Introduction

Modern multi-core and many-core processors provide large computational capacity while keeping the power consumption in check. They can execute thousands of arithmetic instructions in a time required for one random access to the Dynamic Random Access Memory (DRAM) [1], [2]. As a result, lightweight computations frequently stall the cores for data. Yet, processor architectures, programming languages (such as C++ [3]), and multithreading technologies (such as OpenMP [4]) typically describe concurrency and computational structure without explicitly exposing the considerable costs associated with data access.

In order to satisfy the requirements of a broad range of algorithms, modern processors [5], [6] allow random access to the DRAM similar to the von Neumann architecture. To amortize the latency cost of the random access, they use a multi-level cache that reduces the data access cost for reused data. The cache system is transparent to the programmers. It does not need special programming semantics. However, it is also difficult to explicitly control in programs. Programmers need to use an algorithm and data layout that presents higher locality to the cache and expect the memory system will optimize data movement so as to maximize bandwidth and minimize latency. In addition to this, the cache system also tracks concurrent reads and writes for coherency of the data across the cores which can increase the cost of the data access if care is not taken to lower such occurrences. Programmers can estimate the I/O cost of a parallel algorithm from its locality, access pattern, concurrency and cache parameters, and reduce it through program transformations. However, the estimation can be far different from the real cost due to lack of cache control and precise knowledge of the details of the cache and memory scheduling operations.

Another way to lower the data access cost is to have concurrent execution contexts [6], [7] on each core. If at least one context has data for its computation at any instant, the core remains busy. This technique hides the latency of the scattered data access. However, each context gets a smaller allocation of the core's resources such as registers, cache, and scratchpad. It necessitates smaller working set size per context to avoid thrashing. A smaller working set size may reduce opportunities for data reuse so that even though latency is hidden by simultaneous scattered accesses, the overall I/O cost may increase due to increased bandwidth requirement.

To address high latency costs, a core can use a hardware prefetcher that speculatively issues load requests before the core demands the data [8], [9]. A more sophisticated prefetcher prediction logic provides a wider coverage of the suitable data access patterns. However, a complex prefetcher also consumes larger chip area and power. Also, if the prefetcher fails to anticipate the desired data access pattern, then it proves ineffective for hiding latency. Modern processors have prefetchers that detect fine-grained sequential and strided access pattern [5], [6]. Therefore, they are useful only in operations that scan through a region in the DRAM. They also provide software prefetch instructions for other data access patterns. However, their overuse may lower the performance since they consume the core's cycles. Due to these restrictions, the fine-grained (speculative or explicit) prefetching may not improve the performance of an algorithm that has a coarse-grained but predictable access pattern.

Since the prefetched data occupies cache space, the cache replacement policy also determines the data access cost. If the replacement policy distinguishes between the single-use streamed data (from a scan operation) and the reused data (from cache blocking), it can prevent eviction of the reused data due to the streaming access [10]. On the other hand, if the programmer's intention is to reuse the streamed data, the same replacement policy becomes the cause of cache thrashing.

- *Anup Zope is with the Center for Advanced Vehicular Systems, Mississippi State University, Starkville, USA*
- *Edward Luke is with the Department of Computer Science and Engineering, Mississippi State University, Starkville, USA*

Manuscript received (Date); revised (Date).

Most cache techniques lower the latency of the data access using a reactive approach that responds to resource requests. This strategy has certain advantages. First, reactive policies respond to the runtime data access demand of a program. Therefore, they are useful when the program cannot plan the data access ahead of time. Second, they automate the movement of the data through the cache. This takes away the burden from the software developers and processor's execution engines. Third, they keep the operational semantics of the processor intact. It allows changes to the reactive policies without disrupting existing programs. The downside of the reactive approach is that the programmer loses control over the hardware's behavior. Performance is optimized through the reorganization of the instructions and data that can trigger a useful response from the reactive machinery. In some situations, such a rearrangement is beneficial, in others, it degrades performance. Since the latency cost of a program is often highly dependent on specific policies employed by these reactive components, it is very difficult to create programs that have portable high performance. Generally, there is not a fixed set of requirements for these reactive systems because advanced reactive features come at costs of power and chip area. Therefore, changes to these features that better optimize these costs are often a component of new architecture designs.

What is needed are models of computation that describe not only concurrency, but also address the real and fundamental constraints that memory systems impose: Memories that have high capacity also have high latency costs and require burst data transfers to achieve maximum bandwidth, while high speed memories that are close to cores are relatively small and require careful staging to avoid thrashing. What is needed is support for prescriptive methods that allow software to convey the data access requirements of a computation to the hardware such that efficient scheduling of latency-amortized and predictable-cost data access can be achieved. Ideally, these prescriptive methods should have low hardware overhead in that effective scheduling can be performed with minimal control logic, chip area and power consumption. In this paper, we provide a notional hardware model that supports a form of prescriptive access pattern for algorithm development. It is loosely based on the general concept of streaming computations, but is developed with the goal of describing a contract between the hardware and software developers that would facilitate development of performance portable software. We present the model as layers of four levels, level-0 to level-3. Each additional level has enhanced features compared to the previous levels. In a subsequent paper, henceforth called PART-2 [1], we present applications of the model to algorithms of common interest such as merge sort, sparse matrix - dense vector multiplication, and the MapReduce [11] programming model, and present implications of the cost analysis on the design of processor and memory system.

2 THE QUEUE STREAMING MODEL

We made several observations that provide the motivation for the definition of the queue streaming model (QSM). First, it is a well known fact that bandwidth is improving much faster compared to latency. Any improvement in the latency implies an improvement in the bandwidth [1], [12]. Second, the DRAM provides the highest efficiency, in terms of both the power and time, when a computation fetches the entire row of a bank. Therefore, block-by-block streaming access, where a block is equal to the rows of the concurrent DRAM banks, has the lowest cost. Third, the streamed data occupies the cache space: If a computation performs scattered access only to the data stored in the cache, it incurs lower access cost. Fourth, using the cache for write-only data wastes a precious resource. Therefore, it is necessary to stream the write-only data directly to the DRAM to save cache space. It is also necessary to write the data to the DRAM in bursts to amortize the cost of initiating a DRAM transaction. Fifth, a read-modify-write operation on a multi-core processor reduces the number of DRAM accesses, but increases the data access cost due to the expensive read-for-ownership (RFO) cache coherency broadcast messages [5], [13]. Compared to that, the read-only data access is inexpensive. Finally, even though the cache coherency allows the cores on a multi-core processor to perform fine grained communication, their excessive use is detrimental to the performance due to the coherency protocol. Hence, an optimal schedule tries to lower the occurrences by coarsening the task granularity to a level that minimizes the communication but also avoids cache thrashing [14], [15], [16], [17], [18]. These observations have their roots in the fundamental physical constraints on the shared memory processor and memory system design. To abide by these constraints, the QSM specifies a structure of computation that performs only streaming access, keeps the data of each core independent, and avoids inter-core communication. It also provides essential characteristics of the hardware to realize the computation. Following four subsections incrementally describe the model.

2.1 The Level-0 QSM

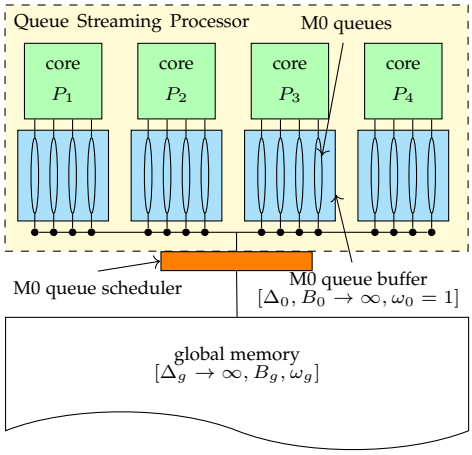

Fig. 1. Schematic of the level-0 queue streaming processor.

The level-0 queue streaming processor (QSP) has P identical *cores* which can compute concurrently and asynchronously. The processor is connected to a *global memory* of large capacity ($\Delta_g \to \infty$) through a *queue scheduler*. Each core has its own set of *queues* that allow it to access data

1. A. Zope and E. Luke, "Queue Streaming Model, Part 2: Algorithms," Submitted for review to TPDS

in the global memory. The queues issue the data access requests to the scheduler according to the demand from the cores. The queue scheduler serializes the data access requests from the queues of all the cores. The queues are also called as C_g-queues since they allow a computation to access the global memory.

The global memory stores data in *blocks* of size ω_g words, which is also one unit of the data transfer between the queue scheduler and the global memory. The reads and writes to the global memory are *symmetric*, that is, they take the same amount of time.

The number of queues on each core is N. A core needs to configure a queue to get strictly sequential access to a contiguous region in the global memory, called *access region* of the queue. The limitations are that the access region of any two queues on any core cannot overlap at any instant of time, and a queue can be used either for reading or writing to the access region. Once configured, the core needs to activate the queue to start the data transfer and deactivate it to stop the data transfer.

The queues serve three purposes. First, a queue provides fine-grained streaming access to the access region irrespective of the block size of the global memory. Each queue tracks current position in the access region, which points to the beginning of the region immediately after configuration. Any operation on a queue takes effect with respect to the current position at the time of the invocation. A read queue provides *advance(w)* operation that reads *w* words from the current position in the global memory and advances it by *w* words. A write queue provides *push(w,t)* operation that writes *w* words contained in *t* at the current position in the global memory and advances it by *w* words. A queue can be configured for either forward or backward access to the access region. Second, they hide the latency of the global memory data access. Each core has Δ_0 words of *queue buffer* that the queues on the core share. A read queue uses it to store blocks prefetched from the global memory and supplies the data to the core at word-level granularity ($\omega_0 = 1$) as per its demand. A write queue uses it to accumulate the data generated by the core and eventually writes it to the global memory in block-level granularity. Third, a read queue provides look-back access to the data in the global memory using *get(lb,w)* operation, which supplies *w* words to the core starting from the offset of *lb* words before the current position in the access region. Each read queue uses an additional portion of the queue buffer than that is necessary to hide the read latency to supply the look-back data to the core. The look-back distance available to a core is finite due to the finite capacity of the queue buffer. The cost of accessing the data from the queue buffer is low and uniform ($B_0 \to \infty$).

After activation of a read queue, it takes some time to fill the buffer from the global memory before it can start supplying the data to the core. After deactivation of a write queue, it takes some time to empty out the buffer content to the global memory. Therefore, the activation and deactivation incurs an *overhead* due to the time required to fill or empty the buffer. The overhead is o_g units of time. See Section 3.2 for the estimation of the overhead.

The queue scheduler and the global memory provide bandwidth of B_g words per unit time (called *system bandwidth*). An individual core can achieve maximum bandwidth of B_c words per unit time (called *core bandwidth*) such that $B_c \leq B_g \leq P \times B_c$. In other words, the cores and the queue scheduler have sufficient resources to saturate the system bandwidth when all the cores are active but not necessarily when only a single core is active. Therefore, the *effective bandwidth* when $p \leq P$ cores are active is $B_p = \min(B_g, p \cdot B_c)$.

The queue scheduler uses fair scheduling to serve all the queues on the processor such that a high bandwidth computation on a core does not disproportionately slow down a low bandwidth computation on another core. It means that the queue scheduler serves a request from a queue with bounded latency, irrespective of the number of active queues on the entire processor. Better the worst case bound, smaller is the queue buffer size necessary to make the access latency-free (see Section 3.1).

Computation on this processor proceeds in a series of steps called C_g-steps. A C_g-step is represented by a *queue streaming kernel*. All or a subset of the cores execute the kernel in following three phases:

1) *Initialization*. Each core configures and activates required number of queues.
2) *Computation*. They advance the read queues, read data from their look-back, perform computations on it and push the results to the global memory through the write queues.
3) *Finalization*. Each core deactivates all the active queues.

The cores synchronize when all the C_g-queues deactivate, that is, when the C_g-step finalizes. This mechanism provides *implicit synchronization* since the cores do not need to communicate to synchronize. At a coarser level, this model is similar to the BSP model [19] except that the cores do not communicate with each other. The time slab between the start and end of execution of a kernel is similar to a superstep of the BSP model without the communication.

The computation phase could be viewed as a stream of instruction *windows*. Each window represents an independent computation that contains a mix of the queue advance and push operations that initiate interaction with the global memory, and the arithmetic instructions (including the queue look-back operation) that don't interact with the global memory. Since the instructions are also treated as a stream, a core can use a queue to fetch the instructions.

The level-0 model requires that the assignment of the windows to the cores is determined ahead of the kernel execution (static schedule of execution) since the queue activation and deactivation occurs only in the initialization and finalization phase, respectively.

2.1.1 Estimation of the Execution Time

If we consider the average of the ratio of the arithmetic instructions to the queue operations as a function of the window size, the smallest window size that reflects the bulk computational intensity of the instruction stream is termed as *sustainable window* of the computation. If we denote by C and D the average amount of time spent in executing the arithmetic instructions and the average amount of global

memory data accessed in the sustainable window, respectively, the ratio $\frac{C}{D}$ represents the *sustainable computational intensity* of the computation. If p cores participate in the execution of the kernel, $\frac{pD}{C} < B_p$ implies that the kernel is *compute-bounded*, otherwise it is *bandwidth-bounded*. If V is the total data read and written to the global memory by the queues on all the cores, the execution time of the kernel is given by Equation 1. The overhead is amortized if V is large.

$$t = \max\left(\frac{C}{pD}, \frac{1}{B_p}\right) \times V + o_g \tag{1}$$

2.1.2 Estimation of the Look-back Distance

Since the queue buffer has limited capacity, the look-back distance available to each read queue is finite. It is necessary to know the look-back distance for determining the kernel computation schedule. Here, we outline a method for its estimation.

The queues use part of the queue buffer, let's say $\Delta_{0,\text{latency}}$ words, to store prefetched or accumulated blocks that allows them to hide the memory access latency. The read queues use the remaining part of the buffer to provide look-back access. Section 3.1 provides the estimate for $\Delta_{0,\text{latency}}$, that depends on the queue scheduler service time, the *burst size* of each queue, the system bandwidth, and the block size of the global memory. Since all these factors are known in advance of the kernel computation, $\Delta_{0,\text{latency}}$ can be estimated. From this, the space available for the look-back access is determined as $\Delta_{0,\text{look-back}} = \Delta_0 - \Delta_{0,\text{look-back}}$. This space is distributed to the read queues in proportion to the sizes of their data types. This allows us to determine the look-back distance of each read queue in terms of the number of values of the queue's type.

2.2 The Level-1 QSM

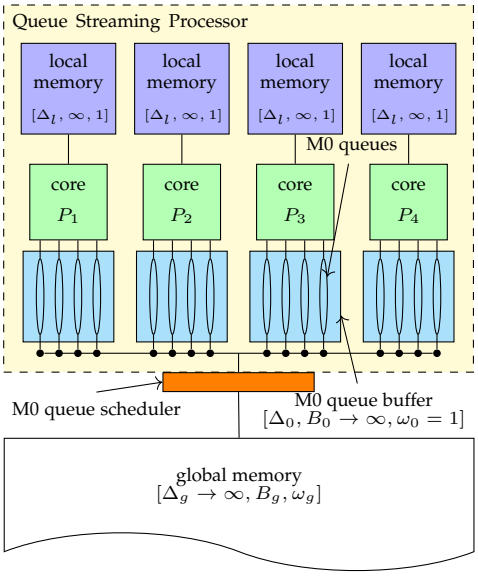

Fig. 2. Schematic of the level-1 queue streaming processor.

In addition to the functionality of the level-0 model, the level-1 model provides a local memory of Δ_l words to each core. Its access granularity is one word ($\omega_l = 1$). The cost of accessing this memory is low and uniform ($B_l \to \infty$). The core explicitly controls the placement and eviction of data in this memory. It can use the memory to accumulate intermediate results of a kernel computation or to store temporary variables. Since the level-1 model functions similar to the level-0 model, the execution time analysis, the requirement of the queue buffer size and the queue scheduler are the same as the level-0 model.

2.3 The Level-2 QSM

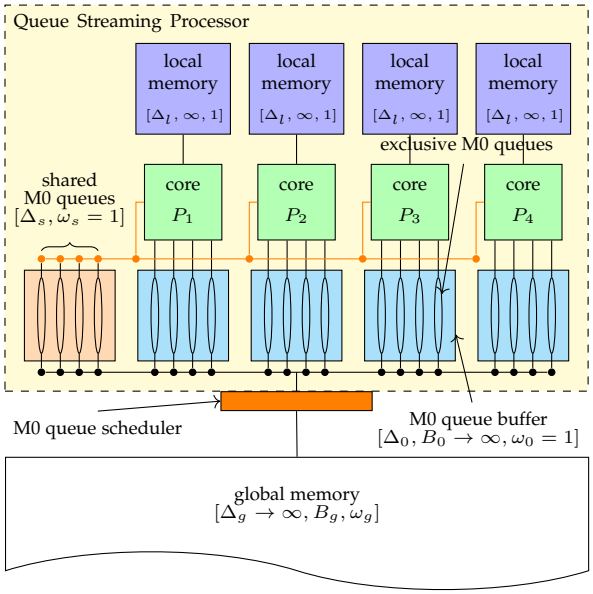

Fig. 3. Schematic of the level-2 queue streaming processor.

The level-0 and level-1 models assume that it is possible to determine a sustainable window for a kernel that represents the bulk computational characteristic of the computation (e.g. the computational intensity) so that a load balancing static schedule can be determined. However, the kernels for which this is not possible, e.g. the kernels for which the computation time of the windows is unknown or has large variance, a static schedule can produce large load imbalance. In this case, it is necessary that the cores dynamically distribute the windows among themselves while the kernel is running. The level-2 model provides N_s shared queues, called $C_{g,s}$-queues. The cores can use them to employ centralized or distributed queue work scheduling, provided the region of the global memory accessed by each window in the instruction stream is determinate and the windows are data independent. The queues provide synchronized, read-only, and sequential access to the data in the global memory. The cost of the synchronous access to a shared queue is o_s units of time.

The structure of a C_g-step of the level-2 model is similar to the level-0 and level-1 models, except a few differences. While the model continues to support activation and deactivation of the exclusive queues (C_g-queues) in the initialization and finalization phases, respectively, it also allows it to happen in the computation phase. However, a kernel can activate and deactivate the shared queues ($C_{g,s}$-queues) only in the initialization and finalization phases,

respectively. Cores synchronize when all the shared and exclusive queues on the processor deactivate. Therefore, the finalization phase initiates barrier synchronization between the cores. In addition to this, each core has two execution contexts (virtual cores) that equally divide the per core resources such as the queues, queue buffer, and the local memory among them. The overhead of a context switch is o_v units of time.

Each core uses the C_g-queues to access data in the global memory. In the level-0 and level-1 model, the queues are configured in the initialization phase since the static schedule provides the extents of the access region for each queue before the computation starts. In case of a dynamic schedule, the extents are known, but the assignment of the windows to the cores is unknown. Therefore, a dynamic schedule requires cores capable of configuring, activating, and deactivating the C_g-queues during the computation phase while the extents for the configuration are read from the $C_{g,s}$-queues that are configured in the initialization phase.

While the dynamic scheduling can use a much finer granularity (e.g. window level granularity) for better load balancing, it results into excessive overhead from the synchronized access to the $C_{g,s}$-queues and the activation/deactivation of the C_g-queues (combined overhead of $o_s + o_g$) during the computation phase. Therefore, the dynamic schedule has to trade-off load balance with the overhead by agglomerating consecutive windows into chunks. Even with an optimal trade-off, the computation has to bear the overhead. The two execution contexts allow enough parallel slackness for overlapping the overhead with the computation. A perfect overlap requires that each chunk has enough computational cost to hide the overhead. That is, $C_{\text{chunk}} + o_v \gtrsim o_s + o_g$. Though more than two contexts provide better parallel slackness, they further divide the resources on the core (such as the queues, queue buffer) that are already limited due to the limited chip area.

A centralized work-queue scheduler requires only a single $C_{g,s}$-queue for all the cores. Each core picks next available chunk from the queue, configures and activates required C_g-queues, performs computations, deactivates the C_g-queues, and repeats the process until the $C_{g,s}$-queue is empty. A distributed work-queue scheduler requires one $C_{g,s}$-queue per core. Each core executes the chunks from its own shared queue until it is empty. After that, it tries to steal chunks from the other cores until there are no chunks left in any of the shared queues.

The execution time of the kernel is determinate if the computation time of each window is determinate. Since a dynamic schedule is used when the kernel is compute-bounded, the computational cost of a chunk i is given by, $C_{\text{chunk},i} = \sum_{k=i_s}^{i_e} C_k$ units of time, where the chunk contains windows in the range $[i_s, i_e]$. The time required to execute the chunk is given by, $t_{\text{chunk},i} = \max\left((C_{\text{chunk},i} + o_v), (o_g + o_s)\right)$. The total execution time of a kernel on a level-2 model can be determined from the particular dynamic scheduling algorithm it uses.

2.4 The Level-3 QSM

A large application that uses the QSM consists of multiple kernels arranged as a directed acyclic graph (DAG). The edges in the graph represent data dependency between the kernels. Any topological order of the nodes in the DAG gives execution schedule of the application for a QSP. However, the schedule may require a large amount of data transfer between the queue buffer and the global memory due to the producer-consumer relationship between the kernels. Fusion of the kernels can lower the cost by eliminating storage of the intermediate streams in the global memory provided the processor has a high bandwidth, large capacity memory (larger than the local memory) for their storage. Even if the memory does not have capacity large enough to store the intermediate streams, the computation can be staged in blocks (by partitioning the input and output streams of the fused kernel) to accommodate for the limited capacity.

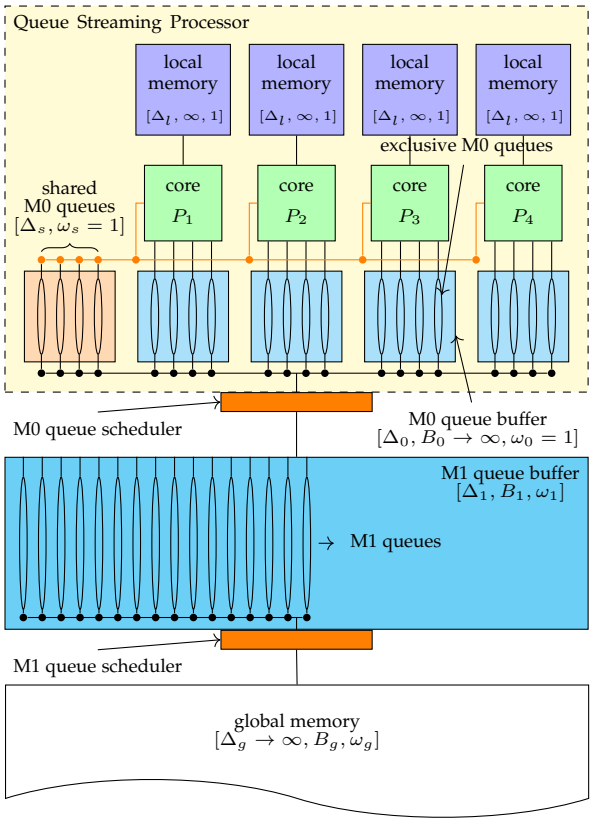

Fig. 4. Schematic of the level-3 queue streaming processor.

For this purpose, the level-3 model provides an additional layer of queues, queue buffer and queue scheduler as shown in Figure 4. The M1 buffer has capacity of Δ_1 words, bandwidth of B_1 words per second, and block size of ω_1 words. Usually, $\Delta_0 < \Delta_1 \ll \Delta_g$ and $B_0 > B_1 > B_g$ since the buffer is closer to the cores compared to the global memory but farther than the M0 buffer. A fused kernel would reduce the bandwidth cost if it stores the intermediate data in the M1 buffer instead of the global memory.

An M0 queue (shared/exclusive) accesses data in the M1 buffer through the M0 queue scheduler. Similarly, an M1 queue accesses the data in the global memory through the M1 queue scheduler. An M0 queue can access data in the global memory by cascading with an M1 queue. If an exclusive M0 queue is configured to access data in the global

memory, it is called as a C_g-queue. If it is configured to access data in the M1 buffer, it is called as a C_1-queue. Similarly, if a shared M0 queue is configured to access data in the global memory, it is called as $C_{g,s}$-queue. If it is configured to access data in the M1 buffer, it is called as $C_{1,s}$-queue. In this text, $C_{1\{,s\}}$-queues refers to the C_1-queues and the $C_{1,s}$-queues, and $C_{g\{,s\}}$-queues refers to the C_g-queues and the $C_{g,s}$-queues.

The level-3 model expands the computational structure of the level-0 and level-2 model. It consists of one or more C_1-steps nested in a C_g-step. Each step has the initialization, computation and finalization phases. The initialization and finalization phase of the C_g-step activate and deactivate $C_{g\{,s\}}$-queues, and that of a C_1-step activate and deactivate $C_{1\{,s\}}$-queues, respectively. The cores synchronize when the C_g-step finalizes, that is, when the $C_{g\{,s\}}$-queues deactivate. They also synchronize when a C_1-step finalizes, that is, when the $C_{1\{,s\}}$-queues deactivate. The overhead of activation and deactivation of the $C_{1\{,s\}}$-queues is o_1 and that of the $C_{g\{,s\}}$-queues is o_g units of time. The overhead of synchronized access to a $C_{1,s}$-queue or a $C_{g,s}$-queue is o_s units of time. The level-3 model imposes the restriction that all the $C_{1\{,s\}}$-queues deactivate before all the $C_{g\{,s\}}$-queues deactivate. The restriction automatically nests the C_1-steps inside the C_g-step. The nested structure also implies that the C_1-steps can use the $C_{g\{,s\}}$-queues. This computational structure is similar to the hierarchical computation structure of the Multi-BSP [20] model.

For the M1 buffer, $\omega_1 < \omega_g$. That is, the M1 buffer has a smaller block size compared to the global memory. An advantage of this is that the M0 buffer requires a smaller capacity to make the data access latency-free since it depends on the block size of the next level of memory, that is, the block size of the M1 buffer in this case. Also, the M1 buffer can tolerate the larger block size of the global memory since it has a larger capacity compared to the M0 buffer.

The execution time of a level-3 queue streaming kernel is determined by summing the execution time of the C_g-step and the nested C_1-steps. The cost analysis should use appropriate bandwidth for the C_1-steps. If a C_1-step uses only the C_1-queues, the effective bandwidth is $B_p = \min(B_1, p \cdot B_c)$. If it also uses one or more C_g-queues, the effective bandwidth is $B_p = \min(B_g, p \cdot B_c)$.

The M0 buffer space required to make a C_1-queue data access latency-free is determined using the block size and bandwidth of the M1 buffer, and the worst case service time of the M0 queue scheduler. However, for a C_g-queue, it is determined using the block size of the M1 buffer, the worst case service time of the M1 queue scheduler, and the bandwidth of the global memory. From these estimates, the look-back available to a computation can be determined. In contrast to the M0 buffer queues, the M1 buffer queues do not use the additional space for the look-back. Instead, a computation can use it to store intermediate results.

3 Management of the Queued Access in the QSM

Unlike the scattered access, which may incur latency for each individual access, the queued access of the QSM incurs the overhead once. Then each access from the queues is latency-free. In other words, we can also say that the queues amortize the overhead. In this section, we will highlight the characteristics of the queue, queue buffer, and queue scheduler necessary to achieve the latency-free data access and also show way to estimate the look-back distance and the overhead.

3.1 Queue Scheduler Policy and Queue Buffer Capacity

One of the purposes of the queues is to provide latency-free access to the data in the global memory. Since they work independently of one another, they simultaneously generate data access requests. However, the queue scheduler serializes the requests because the link between the queue scheduler and the global memory can serve only one request at a time. The queues require sufficiently large buffer to hide the data access latency. The required size of the buffer, in turn, depends on how promptly the queue scheduler can serve the requests. Thus, it is related to the scheduling algorithm used by the queue scheduler. In this section, we highlight the relationship between the two and specify the characteristic of the queue scheduler essential to hide the latency.

To hide the memory access latency, the queues need to saturate the link between the global memory and the queue scheduler. This happens when the aggregate rate of data access by the cores is greater than or equal to the memory throughput. Therefore, for this analysis, we assume that the computation is bandwidth-bounded. We use the level-0 model for this analysis which has only two levels of memories - the M0 buffer and the global memory. However, the results are expandable to the level-3 model with more than two levels of memories.

A core uses *advance(w)* or *push(w,t)* operation of a queue that triggers access to the global memory. These operations cause instantaneous surge in the data access demand. While these instructions for the queue could be distributed throughout the sustainable window, the worst surge happens when they are clustered at the beginning or end of the window. Let $D_{i,\max};\ i = 1, ..., N$, where N is the number of queues on a single core, be the total data access demand of all the clustered requests of the queue q_i, also called as *burst size* of the queue. We will use this worst-case limit to analyze the buffer size of the queue required to hide the data access latency. Let's assume that the queue q_i uses $n_i \geq 1$ blocks (=$n_i\omega_g$ words) of space in the queue buffer on the core to hide the latency. Let's assume that each queue initiates data access request as soon as the buffer has enough space to perform at least one block of data access. This would happen when the core calls *advance(w)* or *push(w,t)* operation on the queue. Also, the number of blocks it would request is at most $\left\lceil \frac{D_{i,\max}}{\omega_g} \right\rceil$. At this point, the amount of buffer space available to the queue for further processing is at least $\left(n_i - \left\lceil \frac{D_{i,\max}}{\omega_g} \right\rceil\right)\omega_g$ words. The maximum number of windows that can be executed using this data is $\alpha_i = \left\lfloor \frac{\left(n_i - \left\lceil \frac{D_{i,\max}}{\omega_g} \right\rceil\right)\omega_g}{D_{i,\max}} \right\rfloor$. Since the kernel is bandwidth-bounded, the minimum time required to compute a window is $\frac{pD_{\min}}{B_p}$ on p cores. Here, $D_{\min}$ is the minimum amount of data accessed by any window of the kernel. Therefore, for

the queue q_i, the time required to fetch $\left\lceil \frac{D_{i,\max}}{\omega_g} \right\rceil$ blocks must be less than the time required to compute α_i windows. Let's say that this time is τ_i. Therefore, we get following relation.

$$\begin{aligned}
\left\lfloor \frac{\left(n_i - \left\lceil \frac{D_{i,\max}}{\omega_g} \right\rceil\right)\omega_g}{D_{i,\max}} \right\rfloor \frac{pD_{\min}}{B_p} &\geq \tau_i \\
\left\lfloor \frac{\left(n_i - \left\lceil \frac{D_{i,\max}}{\omega_g} \right\rceil\right)\omega_g}{D_{i,\max}} \right\rfloor &\geq \tau_i \frac{B_p}{pD_{\min}} \\
\frac{\left(n_i - \left\lceil \frac{D_{i,\max}}{\omega_g} \right\rceil\right)\omega_g}{D_{i,\max}} &\geq \tau_i \frac{B_p}{pD_{\min}} + 1 \\
n_i \geq \tau_i \frac{B_p}{pD_{\min}} \frac{D_{i,\max}}{\omega_g} + \frac{D_{i,\max}}{\omega_g} + \left\lceil \frac{D_{i,\max}}{\omega_g} \right\rceil & \qquad (2)
\end{aligned}$$

Hence, the least buffer size (in terms of the number of blocks) required for the queue i to hide the latency is given by Equation 3.

$$n_i = \left\lceil \tau_i \frac{B_p}{pD_{\min}} \frac{D_{i,\max}}{\omega_g} + \frac{D_{i,\max}}{\omega_g} + \left\lceil \frac{D_{i,\max}}{\omega_g} \right\rceil \right\rceil \qquad (3)$$

This shows that the buffer size required for each queue to hide the latency is bounded as long as the time required to serve the $\left\lceil \frac{D_{i,\max}}{\omega_g} \right\rceil$ blocks of data from the global memory is bounded. This makes the schedules such as those belonging to the class of round robin (RR) [21], virtual clock [22], fair queuing (FQ) [23], [24] etc. suitable candidates for the queue scheduler policy, but not a schedule such as first-come-first-serve (FCFS) since it may provide unbounded service time to the queues that make infrequent requests. Note that the first-ready first-come-first-serve (FR-FCFS) [25] DRAM specific schedule reduces to simple FCFS schedule for this model since one block in this model is the same as a row buffer of the DRAM. Better the worst-case bound for τ_i, smaller is the buffer size requirement. For a homogeneous kernel, this relation reduces to Equation 4.

$$n_i = \left\lceil \tau_i \frac{B_p}{pD} \frac{D_i}{\omega_g} + \frac{D_i}{\omega_g} + \left\lceil \frac{D_i}{\omega_g} \right\rceil \right\rceil \qquad (4)$$

For simple round-robin scheduler, τ_i is given by Equation 5.

$$\tau_i = p \frac{\omega_g}{B_p} \sum_{j=1}^{N} \left\lceil \frac{D_j}{\omega_g} \right\rceil \qquad (5)$$

In this case, the queue buffer size on a single core required to hide the latency is given by,

$$\begin{aligned}
&\Delta_{0,\text{latency}} \\
&= \omega_g \sum_{i=1}^{N} n_i \\
&= \omega_g \left(\sum_{i=1}^{N} \left\lceil \frac{1}{D} \sum_{j=1}^{N} \left\lceil \frac{D_j}{\omega_g} \right\rceil D_i + \frac{D_i}{\omega_g} + \left\lceil \frac{D_i}{\omega_g} \right\rceil \right\rceil \right) \\
&< \omega_g \left(\sum_{i=1}^{N} \left(\frac{1}{D} \sum_{j=1}^{N} \left\lceil \frac{D_j}{\omega_g} \right\rceil D_i + \frac{D_i}{\omega_g} + \left\lceil \frac{D_i}{\omega_g} \right\rceil + 1 \right) \right) \\
&< \omega_g \left(2 \sum_{j=1}^{N} \left\lceil \frac{D_j}{\omega_g} \right\rceil + \frac{D}{\omega_g} + N \right) \qquad (6)
\end{aligned}$$

In general, Equation 7 gives the total queue buffer size. The read queues use the excess space of $\Delta_{0,\text{look-back}} = \Delta_0 - \Delta_{0,\text{latency}}$ words for the look-back access. It is distributed among them in proportion to the size of their value types so that they can provide the same look-back distance (in terms of the number of instances of the value type) to the kernel computation.

$$\Delta_0 = \Delta_{0,\text{latency}} + \Delta_{0,\text{look-back}} \qquad (7)$$

3.2 Estimation of the Queue Overhead

The cost analysis of a queue streaming computation takes into account the overhead of activation and deactivation of C_g- and C_1-queues. The overhead of a read queue is due to the time required to fill the M0 queue buffer before it can start supplying the data to the core. For a write queue, it is due to the time required to empty the buffer content. The overhead of a C_g-step is o_g, and that of a C_1-step is o_1. The overhead depends on the number of read and write queues used by the computation per core, the burst size of each queue, and the number of cores that take part in the computation.

Let's assume that $Q_{r,g}$ is the set of read queues and $Q_{w,g}$ is the set of write queues of a C_g-step. As described in Section 3.1, a read queue $q_i \in Q_{r,g}$ requests $\left\lceil \frac{D_{i,\max}}{\omega_g} \right\rceil$ blocks of size ω_g words each time it has enough space in the M0 or M1 buffer to store the fetched data. After activation of the read queues, the cores cannot begin the computation until each read queue on each core has obtained at least the data of the first request from the queue scheduler. The cores also halt after the finalization phase until data in all the C_g-write-queues is flushed to the memory. Each C_g-write-queue $q_i \in Q_{w,g}$ uses n_i blocks of the M0/M1 buffer space. Let's assume that p cores take part in the computation. Then, the total overhead is given by Equation 8.

$$o_g = \frac{p \cdot \omega_g \left(\sum_{\{i | q_i \in Q_{r,g}\}} \left\lceil \frac{D_{i,\max}}{\omega_g} \right\rceil + \sum_{\{i | q_i \in Q_{w,g}\}} n_i \right)}{\min(B_g, p \cdot B_c)} \qquad (8)$$

Similarly, the overhead of a C_1-step is given by Equation 9. Here, $Q_{r,1}$ and $Q_{w,1}$ are the sets of C_1 read and write queues, respectively.

$$o_1 = \frac{p \cdot \omega_1 \left(\sum_{\{i | q_i \in Q_{r,1}\}} \left\lceil \frac{D_{i,\max}}{\omega_1} \right\rceil + \sum_{\{i | q_i \in Q_{w,1}\}} n_i \right)}{\min(B_1, p \cdot B_c)} \qquad (9)$$

4 Related Work

There are primarily two hardware solutions to improve performance of computations - cache-based multi-core or many-core processors and multi-context processors. Both of these approaches require certain algorithmic modifications to improve performance, that take into account the characteristics of the hardware as well as the algorithm.

A hierarchical cache reduces the effective latency of data access. Applications need to use cache blocking to benefit from the cache. It involves progressive partitioning or agglomeration of the computational tasks into chunks such that the working set of each chunk fits in some level of the cache. Larger than the optimal chunk size causes cache thrashing, increases the effective latency, and degrades performance. Smaller than the optimal chunk size also degrades performance due to the increase in the chunking overhead that results from chunk scheduling [16], duplication of the data and computation at the chunk boundaries [26], read-after-write coherency traffic between the core private caches [16]. The amount of data movement across different levels of the memory hierarchy (a.k.a. cache complexity) is used to measure the effectiveness of a cache blocking strategy since lower cache complexity implies better execution time. Therefore, many models of computation focus on determining cache complexity of algorithms on a cache-based processor. They differ in the type of assumptions they make about the operation of the cache.

The red-blue pebble game [27] targets architectures that have a limited-capacity fast memory, an unlimited-capacity slow memory, and a single processing element. The external memory model [28] considers that the data transfers between the memories in blocks. The parallel disk model [29], [30] considers parallel access to the external memory units by either a single processor or network-connected processors. Memory hierarchy game (MHG) [31] extends the red-blue pebble model [27] to unicore processors with multiple levels of cache. The model assumes blocked data transfer between the memories. There are other models that target multi-level cache architectures such as the hierarchical memory model (HMM) [32], the blocked hierarchical memory (BT) model [33]. The BT model distinguishes between streaming access and scattered access. Alpern *et al.* [34] proposed the uniform memory hierarchy (UMH) model that models a unicore processor with a hierarchical memory. It assumes that the memory capacity, block size, and the latency of each memory grows exponentially with its level. Their work points out that, for an architecture with exponentially increasing latency with the level of memory, the constant factors in the complexity analysis matter. They briefly discussed an extension of the UMH for parallel computations. The parallel external memory (PEM) model [35] assumes that the processor contains multiple cores. Each core has a private cache that stores data in blocks of a fixed size. The cores share an external memory. The data transfer between the two levels of memories takes place in blocks. To communicate with one another, the cores need to write and read the data to the external memory. Savage and Zubair [36] proposed a universal multi-core model (UMM) that extend the memory hierarchy game (MHG) of [31] to a multi-core processor with various degrees of cache sharing. The objective of the model is to determine the I/O complexity at each level of the cache. They observed that appropriate blocking of the computation DAG results in better cache complexity. The work-stealing scheduler [14] is suitable for multi-core processor in which each processor has a private cache and a shared external memory. The parallel depth first (PDF) scheduler [15], [37] improves cache performance of fine grained computations on a multi-core processor in which the cores share a cache level. It performs better compared to the work-stealing scheduler on a processor with a shared cache [38]. The CONTROLLED-PDF scheduler [16] improves cache complexity of the PDF schedule for divide and conquer algorithms on a multi-core processor in which each core has a private L1 cache and a shared L2 cache. It partitions the computations into blocks that fit in the L2 cache and then allow the cores to execute each block in parallel using the parallel depth first schedule (PDF). This strategy reduces cache misses at both the private and shared caches. The Multi-BSP model [20] assumes a hierarchy of memory units with varying degree of nesting and sharing between the processing elements. It defines a nested computation structure that follows the memory hierarchy, and an optimality criteria based on cache misses at each memory level. Each level in the computation hierarchy has computation, communication, and synchronization phases similar to the BSP model [19].

These models estimate the bounds on the number of data movements between cache levels under different architectural constraints. Tighter bounds indicate an I/O efficient algorithm, but not necessarily the best algorithm. In a cache hierarchy with exponentially growing latency and capacity with the level of the cache, the constant factors cannot be neglected if the aim of the analysis is to choose the best algorithm.

Precise estimation of running time is essential for planning computation schedule. Though cache complexity estimates the actual cache hits and misses, it does not necessarily reflect the actual running time of a computation. The data access cost of an algorithm depends on a number of hardware parameters (such as cache type, capacity, latency, replacement policy, banked access, coverage of hardware prefetchers, memory controller scheduling policy, DRAM structure, etc.) and interaction of the algorithm with the hardware (such as scattered or streaming access, cache hit rate, bank conflicts, DRAM row buffer hit rate, etc.). For instance, the cost analysis proposed by the models represents true execution time as long as the processor performs only on-demand data access, but when it is capable of speculative prefetching, the cost can differ by one or two orders of magnitude for a given problem size. Therefore, in addition to the I/O volume, it is necessary to pay attention to the nature of data access, and the available hardware features to get an accurate prediction of execution time. For this purpose, a model that specifies not only the memory parameters but also the mechanism of data movement across the memory hierarchy is essential.

The multi-core models (such as Multi-BSP [20] or UMM [36]) assume that the cores share a level of cache. This allows the cores to perform fine-grained concurrent data access and synchronization via the shared cache at a much lower (compared to the DRAM), but non-negligible cost. Though

the models account for the cost and propose schedules that reduce the occurrences of synchronization and concurrent access, the algorithms based on these schedules have to bear the cost. This results in lower performance compared to the peak machine capacity.

Some models assume that placement of data in memory hierarchy is explicit, but it is often not the case since the cache is inherently transparent and reactive. Such a cache requires an optimal replacement policy that is capable of accurately predicting data reuse. In present architectures, the I/O cost depends on specifics of the replacement policy that is sufficiently non-standardized and undocumented that it makes it difficult to predict the execution time of algorithms on real processors or to guarantee portable performance across various processor generations, particularly if near maximum throughput is desired.

Parallel slackness can also be used to reduce the effective latency of the data access. Many modern processors provide concurrent execution contexts on each core that simultaneously generate data access requests [6], [7], [39], [40], [41]. Since the context switch overhead is negligible, if a context stalls for data, another ready context can be quickly scheduled for execution. In contrast to cache-based processors, these architectures aim to lower not only the memory latency, but also the arithmetic latency and any other latencies in the computation by overlapping many concurrent computation paths. This allows a computation to attain the peak computational throughput of the core, provided they use an optimal amount of concurrency. A lower than the optimal concurrency exposes partial latencies to the computation, and degrades performance. A higher than the optimal concurrency reduces the availability of the shared resources (such as registers, scratchpad etc.), which also reduces the performance of the computation due to higher effective latency. In practice, the optimal concurrency depends on the characteristics of the computation and the target processor. There are performance models for these architectures [42], [43], [44], [45], [46], [47]. Volkov [47] highlighted the interaction between the concurrency, memory latency and throughput, arithmetic latency and throughput, and the arithmetic intensity of the computation. The model categorizes the computation as bandwidth bounded or compute-bounded, and shows that for a compute-bounded computation concurrency is primarily required to hide the latency of arithmetic instructions in contrast to a bandwidth-bounded computation where it is primarily required to hide the memory access latency. Their experimental results show that as the arithmetic intensity approaches the threshold between the compute and bandwidth boundedness, the minimum required concurrency increases, sometimes to a level that is not supported by the processor. Using architectural and algorithmic techniques that allow a bandwidth-bounded computation to achieve the peak bandwidth of the memory system is the only way to improve its flop rate, other than fundamentally changing the computation to make it compute-bounded, which is infeasible in many cases.

Though scattered data access gives much convenience for algorithm design, it comes with penalties at both the hardware and software layers. It is not possible to speculatively prefetch data accessed in a complex pattern since the hardware to do so is complex and may take a large amount of chip area or it may not cover all the use cases [8], [9], [48]. Therefore, algorithms that rely on scattered access need to use a cache-based or multi-context processor. Both of these approaches suffer from following issues. First, both of the hardware solutions make hardware design complicated. In a cache-based processor, the cache coherency required to support fine-grained concurrent scattered access is expensive, especially when the number of cores grow. A multi-context processor requires a hardware scheduler for the dynamic scheduling of the contexts. Second, the fine grained scattered access causes low utilization of DRAM row buffers and increases the energy consumption. Finally, both the hardware solutions make it harder to model execution time of algorithms. On a cache-based processor, it requires consideration of cache type, associativity, capacity, latency, and replacement policy (which is non-standard and undocumented), and cache miss rate of the computation. On a multi-context processor, it requires consideration of memory throughput and latency, arithmetic throughput and latency, concurrency, and arithmetic intensity. This makes it harder to determine impact of algorithmic changes on the overall performance, ensure performance portability, or to determine the best algorithm. In contrast to the scattered access, streaming access requires a simple hardware to hide the memory latency, does not require significantly large concurrency, and reduces energy wastage due to higher utilization of DRAM row buffers. Also, it is much easier to reason about algorithmic changes since the performance model requires consideration of only the processor's arithmetic and memory throughput, and the algorithm's arithmetic intensity, and data volume, which are known in advance for many algorithms.

The models and hardware features surveyed in this section are applicable to general computations, but they provide little insight into finer optimizations that are driven by the physical constraints on the nature of data storage (e.g. banked data storage in DRAM) and the fundamental limitations on efficient data access (e.g. utilization of entire DRAM row buffers and streaming access). In contrast to the earlier work, the QSM shows the inter-relationship between the execution time and the design parameters of the memory storage, memory controller, and DRAM (demonstrated in detail in PART-2). Here, the focus is not on modeling existing processor architectures, but on providing theoretical justification for the minimal essential features of a processor necessary to make the data access latency-free. The features have their roots in the fundamental physical constraints on the data storage and access. It also promotes a programming model that abides by these constraints. By its nature, the model allows precise estimation of the execution time of a model-conforming algorithm and gives insight into finer optimization opportunities. For example, the QSM determines the look-back size from the data access rate of each queue and the policy used by the queue scheduler. This estimation allows an algorithm to adapt the data layout that can successfully make the data access latency-free. See the analysis of the sparse matrix-vector multiplication presented in PART-2. It shows that the data access cost is predictable. Therefore, the execution time is also predictable for the bandwidth-bounded computation.

5 Extensions of the QSM

In this section, we describe possible extensions of the QSM that provide additional features on top of the base model features.

5.1 Core Level Parallel Features

The model does not specify the capabilities of the core, but modern systems will be composed of cores that utilize the pipeline and vector (SIMD) parallel execution modes. Both the forms of parallelism will share the M0 buffer, and the instructions will have to be scheduled to exploit this parallelism. A straightforward way to schedule this parallelism is to interleave a number of virtual instruction streams so that sufficient consecutive accesses to memory are guaranteed to be independent. This naive approach may squander the valuable local high-speed memory resource, reducing opportunities for data reuse. In cases where the computation is severely bandwidth limited, it may be more productive to limit the core parallelism in order to increase data reuse. In addition, more sophisticated scheduling policies can be used that can take advantage of the implicit synchronization implied by the core internal parallelism.

In case of the pipeline parallelism for static applications, an offline scheduler can reorder operations in an instruction stream in order to improve pipeline utilization, similar to what would be performed by a dynamic out-of-order execution hardware. When the stream does not have sufficient parallelism to fully hide pipeline latencies, a small number of virtual instruction streams may be interleaved to achieve a balance between computational performance and utilization of the local high-speed memory resources.

For SIMD parallelism, the instruction stream can again be reordered to exploit vector instructions. In this case, each SIMD lane operates on different data items with the same operation, which provides an inexpensive (both in terms of power and chip area) way to boost the computational throughput of a core. On a QSP, the SIMD lanes share the M0 buffer and the queues. That is, the advance and push operations on the queues are performed on behalf of all the SIMD lanes. Theoretically, the lanes can access any data in the local memory and the look-back buffer. However, practical considerations on the design of the memories require a banked access. In this case, the SIMD lanes that access data from the same bank produce conflicts that reduce performance and make the execution time unpredictable if the conflicts are not resolved with a static schedule. In addition to this, the SIMD lanes should provide an inexpensive way to permute the data they own in their registers. A flexible permutation allows the lanes to form producer-consumer relationships between them and avoid the need to access the local memory. Ultimately, such schedules would be able to make better use of the look-back and local memory to compress bandwidth through data reuse.

Ultimately the management of the core internal parallelism can be seen as an issue that is orthogonal to the overall queue streaming strategy: Logically the core computations are independent of one another and cores access main memory through a set of queues.

5.2 Fine Grained Synchronization

The QSM requires that the cores synchronize whenever all the queues that access data from a level of memory (e.g. C_1-queues or C_g-queues) deactivate. A motivation for this arrangement is to avoid expensive cache coherency that is required to simulate a fine-grained synchronization. This makes the hardware simple to design and implement. However, in an advanced schedule, it may be necessary to synchronize a subset of the cores. The QSM can be extended to support the fine-grained synchronization by allowing computations to group a subset of queues so that whenever all the queues in a group deactivate, all the cores that use them synchronize.

5.3 Multi-level Memories

The level-3 model could be extended by having more than two levels of the queue buffers (with associated queues and queue scheduler), each with a larger capacity and block size compared to the previous level, and corresponding nested steps in the computational structure. This structure would produce a further reduction in the bandwidth cost of an algorithm compared to the level-3 model.

5.4 Emulation on a Cache-based Processor

The proposed programming model, at least without significant software support infrastructure, will not be able to support traditional multi-threaded programming models. As a result, this could present a barrier to adoption. However, the features documented in this model can be simulated on modern cache-based systems by making the hardware prefetchers, cache replacement policies, and memory controller aware of the queue based memory accesses. Such a system would allow for the implementation of bandwidth optimized queue streaming computations without necessarily abandoning existing software. The work in [49] shows some promise in this direction. It indicates that on some Intel processors, the QSM is implicitly supported in a limited form.

6 Conclusion

In this paper, we have described a queue streaming model (QSM) for shared memory multi-core computing, and discussed its extensions. The model allows only block-by-block streaming access to the data in the global memory via the queues to ensure highest energy and time efficiency of the latency-free data access. At the same time, it allows consumption and production of data by the cores at a fine granularity via the queue buffer. Also, the model follows exclusive read and exclusive write (EREW) semantics for performance predictability. On a cursory look, even though the EREW blocked streaming access seems a significant hindrance for algorithm design, the demonstrated applications in PART-2 suggest otherwise. In that paper, we showed that the model is not only applicable to the static structured computations such as merge sort, but also to scattered access computations such as sparse matrix - dense vector multiplication (SpMV), and to computations that require dynamic load balancing such as MapReduce [11]. In that

paper, we also demonstrated various optimization opportunities available to these algorithms depending on the level of the QSM supported by a particular processor. The cost analysis allows us to determine the exact execution time of a model-conforming algorithm on a model-conforming processor, that reflects the contributions from the finer optimizations used by the implementation. This is especially important if the objective of the analysis is to determine the best implementation of an algorithm on a given processor. Such type of analysis is useful for runtime systems that need to estimate the execution time before generating the computation schedule on a real hardware as well as to hardware designers who need to choose design parameters for new architectures. In addition, the hardware features proposed by the model do not preclude the possibility that a multi-core processor can support a queue streaming mode that conforms to the model alongside the traditional cache-based, fine-grained multi-core computing mode for backward compatibility.

The model highlights various aspects of the hardware that is necessary to ensure latency-free, low-cost access to the data in the global memory as well as lays out the computational structure for the software developers to achieve predictable execution time of the algorithms. This type of model provides a road map for hardware vendors to support a stable set of features that is necessary to justify the software investment in runtime and scheduling systems that the shared memory multi-core architectures desperately need.

Finally, since the chip area is a limited resource, saturation in the attainable transistor density will compel processor designers to judiciously use the chip area, which is primarily devoted to arithmetic units, high-speed memory system, and control circuits. The minimal control structure required for the QSM provides larger chip area for the arithmetic units and the memory system. In addition to this, the cost analysis provides hardware designers an insight into the algorithmic implications of the design decisions. Thus, the model serves as a bridge between the hardware designers and the software developers.

Acknowledgments

The authors would like to thank the Center for Advanced Vehicular Systems and the High Performance Computing Collaboratory of the Mississippi State University for providing the necessary resources. This work was partially supported by the Department of Defense (DoD) High Performance Computing Modernization Program (HPCMP).

References

[1] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, December 1995.

[2] ——, "Memory bandwidth and system balance in HPC systems," Invited talk, Supercomputing 2016, Salt Lake City, Utah, 2016.

[3] (2017) ISO International Standard ISO/IEC 14882:2017(E) Programming Language C++. Https://isocpp.org/std/the-standard.

[4] OpenMP Architecture Review Board, "OpenMP application program interface version 4.5," November 2015, http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[5] Intel, *Intel ® 64 and IA-32 architectures software developers manual combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*, July 2017.

[6] ——, *Intel® Xeon® Phi ™ coprocessor system software developers guide*, March 2014.

[7] NVIDIA, "CUDA toolkit documentation," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[8] S. Mittal, "A survey of recent prefetching techniques for processor caches," *ACM Computing Surveys (CSUR)*, vol. 49, no. 2, p. 35, 2016.

[9] N. Oren, "A survey of prefetching techniques," in *International Conference on Information and Knowledge and Management-CIKM,-492*. Citeseer, 2000.

[10] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 60–71.

[11] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[12] D. A. Patterson, "Latency lags bandwidth," *Communications of the ACM*, vol. 47, no. 10, pp. 71–75, 2004.

[13] Intel, *Intel ® 64 and IA-32 architectures optimization reference manual*, June 2016.

[14] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

[15] G. E. Blelloch and P. B. Gibbons, "Effectively sharing a cache among threads," in *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2004, pp. 235–244.

[16] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch, "Provably good multicore cache performance for divide-and-conquer algorithms," in *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2008, pp. 501–510.

[17] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[18] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Foundations of Computer Science, 1999. 40th Annual Symposium on*. IEEE, 1999, pp. 285–297.

[19] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[20] ——, "A bridging model for multi-core computing," *Journal of Computer and System Sciences*, vol. 77, no. 1, pp. 154–166, 2011.

[21] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round-robin," *IEEE/ACM Transactions on networking*, vol. 4, no. 3, pp. 375–385, 1996.

[22] L. Zhang, "Virtual clock: A new traffic control algorithm for packet switching networks," in *ACM SIGCOMM Computer Communication Review*, vol. 20, no. 4. ACM, 1990, pp. 19–29.

[23] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," in *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 4. ACM, 1989, pp. 1–12.

[24] J. C. Bennett and H. Zhang, "WF^{2}Q: worst-case fair weighted fair queueing," in *INFOCOM'96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE*, vol. 1. IEEE, 1996, pp. 120–128.

[25] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2. ACM, 2000, pp. 128–138.

[26] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick, "Avoiding communication in sparse matrix computations," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–12.

[27] H. Jia-Wei and H.-T. Kung, "I/o complexity: The red-blue pebble game," in *Proceedings of the thirteenth annual ACM symposium on Theory of computing*. ACM, 1981, pp. 326–333.

[28] A. Aggarwal, J. Vitter *et al.*, "The input/output complexity of sorting and related problems," *Communications of the ACM*, vol. 31, no. 9, pp. 1116–1127, 1988.

[29] J. S. Vitter and E. A. Shriver, "Optimal disk i/o with parallel block transfer," in *Proceedings of the twenty-second annual ACM symposium on Theory of computing*. ACM, 1990, pp. 159–169.

[30] J. S. Vitter, "External memory algorithms," in *European Symposium on Algorithms*. Springer, 1998, pp. 1–25.

[31] J. E. Savage, "Extending the Hong-Kung model to memory hierarchies," in *International Computing and Combinatorics Conference*. Springer, 1995, pp. 270–281.

[32] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir, "A model for hierarchical memory," in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. ACM, 1987, pp. 305–314.

[33] A. Aggarwal, A. K. Chandra, and M. Snir, "Hierarchical memory with block transfer," in *Foundations of Computer Science, 1987., 28th Annual Symposium on*. IEEE, 1987, pp. 204–216.

[34] B. Alpern, L. Carter, E. Feig, and T. Selker, "The uniform memory hierarchy model of computation," *Algorithmica*, vol. 12, no. 2-3, p. 72, 1994.

[35] L. Arge, M. T. Goodrich, M. Nelson, and N. Sitchinava, "Fundamental parallel algorithms for private-cache chip multiprocessors," in *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*. ACM, 2008, pp. 197–206.

[36] J. E. Savage and M. Zubair, "A unified model for multicore architectures," in *Proceedings of the 1st international forum on Next-generation multicore/manycore technologies*. ACM, 2008, p. 9.

[37] G. E. Blelloch, P. B. Gibbons, and Y. Matias, "Provably efficient scheduling for languages with fine-grained parallelism," *Journal of the ACM (JACM)*, vol. 46, no. 2, pp. 281–321, 1999.

[38] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry *et al.*, "Scheduling threads for constructive cache sharing on CMPs," in *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, 2007, pp. 105–115.

[39] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE micro*, vol. 30, no. 2, 2010.

[40] M. Forsell, "Configurable emulated shared memory architecture for general purpose MP-SOCs and NOC regions," in *Proceedings of the 2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*. IEEE Computer Society, 2009, pp. 163–172.

[41] M. Forsell, E. Hansson, C. Kessler, J.-M. Mäkelä, and V. Leppänen, "NUMA computing with hardware and software co-support on configurable emulated shared memory architectures," *International Journal of Networking and Computing*, vol. 4, no. 1, pp. 189–206, 2014.

[42] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 152–163.

[43] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A performance analysis framework for identifying potential benefits in GPGPU applications," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 11–22.

[44] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 2011, pp. 382–393.

[45] J.-C. Huang, J. H. Lee, H. Kim, and H.-H. S. Lee, "Gpumech: GPU performance modeling technique based on interval analysis," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 268–279.

[46] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for GPU architectures," in *ACM Sigplan Notices*, vol. 45, no. 5. ACM, 2010, pp. 105–114.

[47] V. Volkov, "Understanding latency hiding on GPUs," Ph.D. dissertation, UC Berkeley, 2016.

[48] J. Lee, H. Kim, and R. Vuduc, "When prefetching works, when it doesn't, and why," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 1, pp. 2:1–2:29, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2133382.2133384

[49] A. Zope and E. Luke, "A block streaming model for irregular applications," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 753–762.

Anup Zope Anup Zope received his undergraduate degree in mechanical engineering from College of Engineering, Pune, India in 2005 and MS degree in computational engineering from Mississippi State University, USA in 2015. Currently, he is a PhD candidate of computational engineering and graduate research assistant at Center for Advanced Vehicular Systems, Mississippi State University, USA. His research interests include parallel algorithm design and analysis, performance tuning and measurement, computer architecture, and scientific computing. He is a student member of the IEEE.

Edward Luke Edward Luke received a BS degree in electrical engineering from Mississippi State University in 1987 and his MS and Ph.D. in computational engineering from Mississippi State University in 1993 and 1999 respectively. He is currently a professor in the department of computer science and engineering at Mississippi State. He is the developer of the auto-parallelizing Loci framework and the CHEM multi-physics computational fluids solver that is widely used in the aerospace community.