Approximate NoC and Memory Controller Architectures for GPGPU Accelerators

Venkata Yaswanth Raparti, Student Member, IEEE and Sudeep Pasricha, Senior Member, IEEE

Abstract—High interconnect bandwidth is crucial for achieving better performance in many-core GPGPU architectures that execute highly data parallel applications. The parallel warps of threads running on shader cores generate a high volume of read requests to the main memory due to the limited size of data caches at the shader cores. This leads to a scenarios with rapid arrival of an even larger volume of reply data from the DRAM, which creates a bottleneck at memory controllers (MCs) that send reply packets back to the requesting cores over the network-on-chip (NoC). Coping with such high volumes of data requires intelligent memory scheduling and innovative NoC architectures. To mitigate memory bottlenecks in GPGPUs, we first propose a novel approximate memory controller architecture (AMC) that reduces the DRAM latency by opportunistically exploiting row buffer locality and bank level parallelism in memory request scheduling, and leverages approximability of the reply data from the DRAM, to reduce the number of reply packets injected into the NoC. To further realize high throughput and low energy communication in GPGPUs, we propose a low power, approximate NoC architecture (Dapper) that increases the utilization of the available network bandwidth by using single cycle overlay circuits for the reply traffic between MCs and shader cores. Experimental results show that Dapper and AMC together increase NoC throughput by up to 21 percent; and reduce NoC latency by up to 45.5 percent and energy consumed by the NoC and MC by up to 38.3 percent, with minimal impact on output accuracy, compared to state-of-the-art approximate NoC/MC architectures.

Index Terms—GPGPU, approximate computing, network-on-chip, memory controller

1 INTRODUCTION

For today’s high-performance computing workloads, general purpose graphics processing units (GPGPUs) have become highly popular due to their support for massive thread and data level parallelism. GPGPUs typically have many computational units within their streaming multiprocessors (SM) [1] (also known as shader cores) that execute thousands of threads simultaneously. Libraries such as OpenCL [2], and CUDA [3] have enabled programmers to efficiently parallelize a program and reap the best performance out of the available computational resources on a GPGPU. However, parallel applications often generate large volumes of memory requests to memory controllers, resulting in a rapid influx of reply data from DRAM to be transmitted from MCs to SMs, which creates memory bottlenecks. Traditional MCs use first-ready first-come-first-serve (FR-FCFS) that is not designed to handle such requests that may have high row buffer locality (RBL) and bank level parallelism (BLP) simultaneously. Also, traditional mesh based NoC architectures are not designed to handle the high volumes of reply traffic between MCs and cores. To support the high traffic rates of GPGPUs, NoC channel widths should be increased multifold compared to conventional NoCs, but this leads to a significant increase in GPGPU power dissipation. Fig. 1 gives a breakdown of power consumed by various components of a GPGPU when executing data parallel CUDA applications from the CUDA SDK sample code suite. For memory intensive applications that generate high volumes of memory requests, the NoC and MC together dissipate up to 30 percent of the overall chip power.

Thus, there is a need for innovative NoC and MC architectures that can support the high volumes of data generated and consumed in GPGPUs, while dissipating low power (and energy), and without sacrificing application performance.

Recent works [4], [5] have demonstrated the impact of a new paradigm called approximate computing that trades-off computation accuracy for savings in energy consumption. Many emerging applications in the domains of machine learning, image processing, and pattern recognition are today exploring approximate computing techniques to save energy and also improve application performance while tolerating a small range of output errors. One of the main goals of our work is to exploit data approximation intelligently to increase the throughput of data movement between MCs and shader cores (SMs) to speed up application execution and also minimize the energy consumed during application execution on GPGPU platforms.

In a typical many-core GPGPU platform, the NoC traffic consists of load/store (LD/ST) data with LD replies forming a majority of the traffic that causes MC bottlenecks [6]. Typically, the traffic in a NoC is skewed with memory requests that follow a many-to-few pattern from cores to MCs and replies that follow a few-to-many pattern from MCs to cores. With high volume of LD reply data at MCs, a bottleneck is created which leads to high wait times for the thread blocks executing on shader cores, and hence a slowdown in overall performance. Hardware designers have come up with high...
radix NoCs with intelligent routing schemes [7], or complex warp scheduling schemes [8] to minimize the MC bottleneck. However, these techniques incur high power/area overheads.

In this paper, we conjecture that several applications that use GPGPUs generate high volumes of data with redundant or similar values that can be approximated to reduce the number of packets transmitted from MCs to cores. We leverage this observation and propose an approximate MC architecture and a high-speed circuit overlay based NoC architecture that together overcome the MC bottleneck issue and minimize energy consumption more aggressively than state-of-the-art techniques, with minimal application accuracy degradation. Our novel contributions are summarized as follows:

- We introduce a novel approximate memory controller architecture that incorporates approximate data-aware memory scheduling in the request channel, to increase MC throughput by intelligently leveraging row buffer locality and bank level parallelism of DRAM requests; we extend the AMC with support for flagging read reply data arriving from DRAM for potential approximate transmission over the NoC;
- We introduce a data-aware approximate NoC architecture (Dapper) that utilizes asynchronous fast overlay circuits for transmitting both general and approximate data between MCs and cores in 3 cycles; we also design a novel NoC router architecture, and an arbitration mechanism that uses a global overlay manager (GOM) to share the overlay circuits between different MCs and cores;
- We conduct rigorous simulation-based experimentation of our proposed Dapper NoC and AMC architectures for various CUDA applications to compare the performance and energy consumption against the state-of-the-art.

2 RELATED WORK

Several prior works have addressed the issue of NoC throughput and energy consumption for traditional many-core processors. In [9], the authors propose a small world NoC that utilizes machine learning to establish fast connections between cores that generate high volumes of data. In [10], the authors propose an application criticality-aware packet routing scheme that prioritizes memory requests of time-critical applications. In [11], [26], [27], [28] fault-tolerant and energy-aware NoC routing schemes are proposed. In [35], [36] complex-network based application scheduling and mapping mechanisms have been proposed to minimize inter-cluster communication in heterogeneous platforms. These works are orthogonal to the proposed approximate NoC and MC architectures. Further, these techniques perform well in CPU based platforms under lighter traffic conditions, but they cannot be used to minimize the bottleneck caused at MCs in GPGPU platforms.

In [37], [38] heterogeneous NoC architectures have been proposed to address many-to-few traffic patterns in CPU-GPU heterogeneous architectures, but, they have not addressed the bottleneck caused by few-to-many traffic between the last level caches (LLCs) near MCs and the cores. Further, [39] and [40] discuss the benefits of mesh based NoC topologies such as scalability, simplicity and ease of implementation. Hence, in this work we customize the mesh based NoC topology for few-to-many traffic pattern in GPGPU processors. The memory bottleneck issue in many-core CPUs and GPUs has been addressed in a few prior works. In [29] an application-aware staged memory scheduling mechanism is proposed that creates batches of requests to maximize row buffer hits and bank level parallelism (BLP) in the DRAM, and to minimize the inter-application bandwidth interference. In [30], an MC that uses BLP-aware prefetching is proposed, to increase MC and DRAM throughput. But these works ignore the adverse impact of NoC latency on the memory bottleneck issue. In our work, we address the memory bottleneck issue from a perspective of both intelligent memory scheduling and minimizing NoC latency.

In [33] a high performance wireless NoC architecture with short ranged wired links and long-range wireless links is proposed to address NoC congestion due to multicast packets arising from cache coherence traffic. However, this work cannot be directly adapted to GPGPU traffic conditions where the reply path has much longer packets (64 B each) compared to cache coherence traffic. In [34] an encoding scheme along with a deadlock free corridor routing and adaptive flit dropping mechanisms are proposed to improve the throughput and latency of NoC under high multicast traffic conditions. In this work as well, the size of the packets is smaller with lower injection rates compared to GPGPU traffic between memory controllers and cores. This technique has the potential to give some improvement in GPGPU NoC throughput in the reply network with applications of lower memory intensity; however, it incurs high power and area overheads at MCs and NoC routers. Unlike the above two works, DAPPER+AMC is more lightweight and is designed to improve the performance of GPGPUs specifically in the commonly observed many-to-few and few-to-many GPGPU traffic conditions.

Approximate (or inexact) computing has been studied by several researchers to save energy by sacrificing the correctness of the output to an acceptable extent. A few works [19], [20] have demonstrated the use of compiler support to label and treat approximate variables differently from other variables and enabling inexact computing on those variables at the hardware level. Other works have used approximate computing at the circuit level [12], [13] to trade-off accuracy of the logic circuits for shorter critical paths and low operating voltages that save power and area footprint. A few other works have introduced approximate last level caches [14], [15]. In these works, the authors reduce the amount of data
stored in the caches, and increase the cache hit rate by associating tags of multiple approximately similar blocks to the same data line. In [17], the authors use approximate computing with spatial and temporal locality to increase the benefit of instruction reuse in GPGPUs. None of these techniques can be applied for resolving the problems arising in MCs and NoC due to the heavy influx of memory requests that cause MC bottlenecks in GPGPUs.

In [18], the authors have proposed a NoC centric approach to minimize the latency caused by congestion in many-core CPUs by introducing approximate compression and decompression of packets at network interfaces, to reduce the number of flits entering the NoC. This work utilizes a dictionary-based compression/decompression technique that consumes high energy and leads to performance overheads when the network traffic is high. However, this work ignores the challenges with MC scheduling to minimize the memory access latency, in GPGPUs. In contrast to these works, in this article, we provide a holistic solution to the memory bottleneck problem in GPGPUs, with a multi-pronged solution that uses approximate computing design principles with smart resource adaptation at both the MC and NoC architecture levels.

3 BACKGROUND AND MOTIVATION

3.1 Baseline Configuration

We consider a heterogeneous accelerator-based system with an x86 CPU and a grid of shader-cores of GPGPUs that have private L1 caches (instruction and data) to support data parallel multithreaded application execution. A shader core consists of parallel integrated pipelines with a common instruction fetch unit that executes a single instruction on multiple data (SIMD) simultaneously. Each integrated pipeline has an integer arithmetic logic unit and a floating-point unit. A shader core also has several load store units that fetch data from a private L1 cache or from the main memory. A GPGPU based accelerator has an L2 cache that is shared across the shader cores. Each shader core also has a texture cache and a scratchpad memory. All shader cores and MCs are connected to an on-chip interconnection network.

Typically, there is little to no direct communication between the shader cores on the chip, as there is no data shared between shader cores that are executing thread blocks. Hence, the L2 is used to cache the contents of private memory of each shader core. The communication between CPU and GPU cores takes place through main memory. The shader cores send read/write requests (via LD/ST instructions) to MCs over the NoC. A memory reply takes several cycles based on the location and availability of data in the L2 or DRAM. The baseline NoC architecture between the shader cores and the MCs has a channel width of 128-bits, twice the size of 64 bit NoC channels in traditional CPU based platforms, and consists of 4-stage routers (stage 1: buffer write; stage 2: route computation, stage 3: virtual-channel/switch allocation; stage 4: switch/link traversal). There are 5 virtual channels (VCs) per input port and 4 flit buffers for each VC. Flits are routed along the XY path from source to destination.

In this work we focus on optimizing the MC as well as the network between shader-cores and MCs. Henceforth, the term cores implies shader-cores for the remainder of this article.

3.2 Data Value Approximability

Several types of data parallel applications are typically executed on GPGPU platforms. Many of them belong to the domain of image processing, signal processing, pattern recognition, and machine learning. There also exist other scientific and financial applications that use GPGPUs that operate on large input data sets. The data used for executing image and signal processing applications in many cases is highly approximable. For example, as shown in Fig. 2a, the areas in boxes contain pixels that are very similar to their adjacent pixels. The RGB values of two pixels for each box are shown in Fig. 2b. These values (for each box) are quite similar and it is not energy efficient to save and transmit cache lines containing similar RGB values separately from DRAM to the cores. Instead, if cache lines with same (or similar) data can be identified at the MCs, we can avoid their repeated transmissions to the cores, to save energy. Dapper and AMC are designed to realize this idea.

The next logical question one may ask is: how approximable are typical applications that run on GPGPUs? Fig. 3a shows the percentage of approximable data in different parallel CUDA applications, while still generating acceptable results. Applications such as Discrete Cosine Transformation, Convolution Texture, and Raytrace can have up to 78 percent of approximable input data that can be exploited to mitigate the memory bottleneck issue and save NoC energy. By making use of the programming paradigm proposed in [19], it is possible to specify approximable variables using the @approx keyword as highlighted in green in Fig. 3b. Such approximable variables will have a distinct set of instructions for load and store that are used when compiling the C++ code to machine code. These instructions can be used by cores and network interfaces to identify approximable data and accept inexact values for them from main memory.

3.3 Memory Scheduling in GPGPUs

DRAM is organized into a hierarchy of modules as follows: each MC has a dedicated channel to access DRAM DIMMs that each consist of two ranks. Each rank typically has 8 or 16 DRAM chips depending on the data bus width (8 or 16
Each chip has up to 32 banks that activate rows of data to be accessed, which are then buffered in each bank. An MC sends requests to access specific rows and columns within those rows across banks to read/write data. Once accessed, a row is either kept open (to enable fast accesses for future requests to the same row) or closed (written back to its respective bank before a new request arrives). Memory accesses that hit the rows that are already buffered are said to have row buffer locality (RBL). The performance of DRAM is enhanced by scheduling requests intelligently to increase the utilization of rows by maximizing RBL. If the subsequent accesses are not in the same row, they have to wait until the next row is buffered.

To minimize access latency, DRAMs today also support bank level parallelism where accesses to multiple banks are scheduled in parallel. Fig. 4a shows the average RBL of memory requests waiting in the MC input queue at any instance, and Fig. 4b shows the average BLP in memory requests waiting in the MC input queue, for different CUDA applications. Most of the applications have high spatial locality leading to high RBL of around 3. CUDA applications also have a high BLP of above 3 in some applications due to their inherent data parallelism. However, prior work [29] shows that RBL and BLP is not easy to harness at the same time, making designers choose among the two options to increase DRAM performance. In contrast, in AMC we propose a novel credit-based request scheduling mechanism that leverages both RBL and BLP based on the runtime application characteristics, to increase NoC throughput and leverage the spatial locality of the approximable DRAM requests.

3.4 Overlay Circuits for Low Latency Traversal

As discussed earlier, in GPGPUs, read reply data is the main source of MC bottlenecks. Minimizing read reply latency is crucial for application performance. Also, read reply data comprises most of the traffic from MCs to cores in a few-to-many traffic pattern. To ensure that MCs do not get clogged by a heavy influx of reply data, it is intuitive to design a NoC with all-to-all connections. However, power and wire routability constraints in modern chips limit such architectures. Hence, researchers have come up with smart solutions in [6], [7] where they propose intelligent NoC routing schemes in tandem with MC placement to create conflict free paths in NoCs for packets to traverse from MCs to cores. However, the complex router design in such architectures adds to NoC power overheads.

In our work, we make use of the few-to-many traffic pattern of the reply packets and utilize an overlay circuit topology that forms dedicated circuits for each MC. An overlay circuit connects one MC to one core, forming return paths for read reply packets. Fig. 5 shows how the circuits are established from each MC to all the cores, on a 2D mesh-based NoC topology. Each overlay circuit established from an MC to all the cores stays for a fixed time window during which it transmits the reply packets of that MC, before switching to the next overlay circuit (for another MC). On overlay circuits formed between an MC to the cores stays for a fixed time window during which it transmits the reply packets of that MC, before switching to the next overlay circuit (for another MC). On overlay circuits (shown with red and green colored arrows in the figure) each MC transmits flits of packets waiting in its queues that reach their destination cores in 3 cycles (or less) using asynchronous links and repeaters that bypass one or more NoC routers. Flits traverse in X and Y directions in one cycle each, stopping only at the turns. Hence, flits traversing over overlay circuits do not need switch arbitration and route computation at every hop. This leads to a low energy consumption NoC that establishes high-throughput,
congestion-free paths for read reply packets. More details about how overlay circuits are established and used for data traversal are discussed in the following sections.

4 **Overview of AMC and Dapper**

In our approximate memory controller, we enhance GPGPU MC throughput by exploiting the inherent parallelism in request packets, and the approximability of the read reply data. In *Dapper* we reduce the latency of on-chip transfers by optimizing for the few-to-many pattern of the approximable read reply data packets between AMCs and cores.

The *AMC* consists of two key components: (1) Approximate data-aware memory scheduling in the request channel that intelligently switches between BLP or RBL with a credit-based throttling mechanism, and (2) Data approximation support in the reply channel of the memory controller, that identifies approximable data waiting in its output queues and marks them for *coalescence*. The overlay circuits of *Dapper* then broadcast the flits of *coalesced* packets to their destinations over fast overlay circuits that are established for each AMC. We define *coalescence* as grouping of DRAM replies whose data from AMCs are approximable and within an *error_threshold*. These replies are then *coalesced* into a single packet, to reduce the number of packets transmitted into the NoC for delivery to multiple destinations.

*Dapper* employs a 2D-mesh based NoC topology to create overlay circuits between cores and AMCs. The network is divided into two planes (request and reply) each with 64-bit channel width, to avoid protocol deadlock. The NoC routes packets with an XY turn-based routing scheme to avoid routing deadlock. The request packets are transmitted on a request plane and the reply packets are transmitted via fast overlay circuits on the reply plane. A majority of the reply plane traffic consists of read reply packets. Hence, *Dapper* performs packet coalescing using approximation only on read reply data that is received from DRAM. The reply plane of the NoC contains a novel router architecture that enables establishing and tearing down of the overlay circuits. More details of these modules are discussed in the following sub-sections.

4.1 **Approximate Memory Controller (AMC)**

AMCs are connected to the NoC through a network interface (NI) and a router. An AMC receives memory requests from cores and creates commands to send to DRAM. Before sending a request to an AMC, a core sets an *approximable flag* for the memory request, if the load operation is on an approximable variable. The NI that connects the core to the router generates flits from the memory request packet and adds an *approximable flag* to the header flit. The receiver NI connected to the AMC generates a memory request from these flits and sets an *approximable flag* for the memory request before sending to the memory subsystem, as shown in Fig. 6.

4.1.1 **Approximate Data-Aware Memory Request Scheduling**

On the request channel, the AMC has a novel scheduler to intelligently leverage RBL and BLP of the arriving approximable requests. An important observation is that in some applications such as *DCT*, *ConvTex*, and *RAY*, most of the approximable data is located in nearby memory addresses leading to a higher row buffer locality, as shown in Fig. 4a. For these applications, it is preferable to send those requests in a *burst* in consecutive clock cycles so that the replies arrive in consecutive DRAM cycles, and can be stored in adjacent output queues in the AMC, which facilitates higher *packet coalescing* in the output queue. However, it is also preferred to support bank level parallelism to reduce the waiting time of the high volumes of parallel memory requests in GPGPU applications. To leverage both RBL and BLP, we propose a credit-based scheme that prioritizes RBL burst commands when there is a high influx of approximable requests into the AMC’s input queue, and BLP otherwise. To facilitate this, we divide the input queue of an AMC into *n* parallel queues as shown in Fig. 6.

The incoming requests are stored in one of the *n* parallel queues based on the *bank number* of the memory request. An input queue is allocated to each memory request by performing a modulo operation on its bank number with the total number of input queues (*queue_id = bank_number % total_queues*). For each input queue, the scheduler assigns a fixed and equal number of credits at boot up time. For each incoming request processed from a queue, a credit is deducted. The queues are processed only when their available credits are > 0. The total number of credits is equal to the total available input command queue depth at the interface between the AMC and DRAM. The AMC receives a signal from the DRAM after a command is processed internally to replenish the credits back to the queues.

Fig. 6. Overview of approximate data-aware memory scheduling in approximate memory controller.

Algorithm 1 describes the functionality of the scheduler. The algorithm takes number of approximable requests waiting in each queue (*num_approx_reqs*), and *check_depth* as inputs. At every cycle, the scheduler selects a queue and checks if the total number of waiting approximable requests is below a threshold (line 4). It then checks if that queue has at least one credit to process a request (line 5). If both conditions are true, then the input queue does not contain sufficient requests to hit the same row buffer, thus the credit

---

**Figure 6**: Overview of approximate data-aware memory scheduling in approximate memory controller.

---

*Note: The diagram shows the flow of data and control signals in an approximate memory controller, illustrating the interaction between the router, network interface, and memory controller.*

---

**Algorithm 1**: Approximate Memory Request Scheduling

```python
1. Defining variables: 
   - num_approx_reqs: number of approximable requests waiting in each queue
   - check_depth: depth of the input queue

2. For each queue do:
   2.1. Check if the total number of approximable requests in the queue is below a threshold (4) if both conditions are true, then the input queue does not contain sufficient requests to hit the same row buffer, thus the credit

3. Select the queue with the lowest *num_approx_reqs*.

4. Check if the queue has at least one credit to process a request.

5. If both conditions are true, then process the request.

6. Increment the credit counter.

7. Repeat steps 2-6 for all queues.

---

Authorized licensed use limited to: COLORADO STATE UNIVERSITY. Downloaded on February 26,2020 at 22:32:37 UTC from IEEE Xplore. Restrictions apply.
manager selects the BLP scheme for scheduling to leverage parallelism in DRAM commands. In the BLP scheme, an input queue is serviced in a round robin manner in each cycle to schedule the next command (lines 5-9). The RR arbiter sends the queue id as input to the queue mux to select a single request from this queue to create the next DRAM command (line 6), and a credit is deducted (line 7). The RR arbiter then points to the next queue in a round robin manner for the credit manager to process in the next cycle (line 8). If the queue does not have at least one credit, the credit manager invariably selects the next input queue in an RR manner for processing in the next cycle (line 9). However, if the queue has more approximable requests than the threshold (line 10), the credit manager selects the RBL scheme for the next check_depth DRAM commands (lines 11-17). In RBL, the DRAM commands are sent to hit the same DRAM row buffer to get the maximum approximability in the read reply data. Hence, in each cycle the credit manager processes the same input queue till it reaches check_depth to make sure that all the subsequent commands are sent in a burst (lines 11-13). The credit manager notifies the RR arbiter to select the same queue until the burst mode is completed. As a result, the queue mux selects requests from the same queue for check_depth number of times. The queue’s credits are also deducted by check_depth (line 15). This can lead to negative credits for the queue. The queue is not eligible to send any requests until it replenishes its credits back to at least 1. After sending commands in a bursty manner from the same queue, the command manager sends an input to the RR arbiter to select the next queue for processing in the next cycle (line 17). The credits are replenished in a round robin manner when DRAM notifies the AMC that it has completed processing a command (line 20). The credit replenishment stops if a queue receives its quota of credits that are assigned at boot up time. The AMC has an internal staging buffer as shown in Fig. 6 that stores the requests before processing them to DRAM commands. When this queue is full, the scheduler stops processing any memory requests until it is cleared.

4.1.2 Data Approximation in Reply Channel

Whenever a reply is received from the DRAM, it is matched with the corresponding request and saved in the output queue. In the reply channel, the AMC is equipped with a data approximator which parses through each of the read reply data units waiting at the output queue of the AMC, finds the approximable data among the waiting data, and checks if any data can be coalesced before sending to the NI for delivery to the destination cores. The granularity of approximable data waiting in the output queue is an L1 cache line because a read reply from AMC contains a cache line with the address that led to a cache miss. The data approximator tries to coalesce cache lines waiting in queue entries only if each of the variables contained in the cache lines is approximable and contains similar data values in all the cache line entries. Fig. 7 shows an overview of the data approximator. If the output queue has some approximable reply data, the data approximator coalesces them into a single read reply data unit based on their data values to minimize the number of replies injected by the NI into the NoC.

Algorithm 1. Approximate data-aware scheduling

Inputs: num_approx_reqs (for each queue), check_depth

1: next_queue_id ← 0
2: burst_count ← 0
3: for each cycle do
4:   if next_queue_id → num_approx_reqs < Threshold then
5:      // BLP block
6:     if next_queue_id → credits > 0 then
7:        queue_mux_sel ← next_queue_id
8:        next_queue_id ← credits–
9:     end if
10:     next_queue_id → next_queue_id + 1 mod total_num_queues // RR
11:   else // RBL block
12:     if burst_count < check_depth then
13:        queue_mux_sel ← next_queue_id
14:        burst_count ++
15:     else
16:        next_queue_id → credits = credits - check_depth
17:        burst_count ← 0
18:     end if
19: end if
20: increment queue credits by 1 in a RR scheme if notified by DRAM
21: end for

Algorithm 2 describes the steps involved in the data approximation phase in the reply channel. The algorithm takes two parameters (error_threshold, check_depth) as inputs. The check_depth is the same as check_depth in Algorithm 1, because it signifies the number of requests processed in the input queue for sending an approximable burst of commands to DRAM. The data approximator iterates over each entry in the output queue and checks if the data is approximable (line 5). If the data is not approximable, it sends the data unaltered to the NI for delivery (line 6). If the data is approximable, the algorithm iterates through subsequent data entries to find other approximable data (lines 8-9). If an approximable data is found, the difference between the first approximable data and the found data is calculated (lines 10-11). If the difference is within the error_threshold, the found data entry is erased from the output queue, and its destination address is saved (lines 12-13). This iteration process stops once it reaches the check_depth, which also signifies the maximum number of
packets that are coalesced if their data values are within an error_threshold. The list of destination addresses along with the original address is appended to the first reply (line 18). The approximated data and the list of addresses are then sent to the NI (line 20) to generate a packet that is broadcast to the list of destination nodes on the overlay circuits as explained in the following sections. The cleared entries in the output queue are then released for incoming data from DRAM. The data types considered for approximation include both integer and floating-point types. For floating point data, only the mantissa is approximated for a faster computation.

Algorithm 2. Data Approximator

Inputs: error_threshold, check_depth
1: while Output_queue.size() > 0 do
2: \( \delta \leftarrow 1 \)
3: dest_list \( \leftarrow \emptyset \)
4: reply_data \( \leftarrow \) Output_queue.front()
5: if reply_data.approximable = \( = 0 \) then
6: send_to_NI(reply_data)
7: else
8: while \( \delta < \) check_depth do
9: nxt_data \( \leftarrow \) Output_queue.get(\( \delta \))
10: if nxt_data.approximable = \( = 1 \) then // coalesce the data
11: if (reply_data - nxt_data)/reply_data < error_threshold then
12: dest_list.add(nxt_data.dest)
13: Output_queue.erase(\( \delta \))
14: end if
15: end if
16: \( \delta \)++
17: end
18: reply_data.add_dest(dest_list)
19: end if
20: send_to_NI(reply_data)
21: Output_queue.pop()
22: end while

4.1.3 Overheads
The power and area overheads involved in approximate data-aware scheduling in the request plane, and data approximation in the reply plane are minimal compared the power consumed by the computing shader-cores. Algorithm 1 needs \( n \) additional counters to keep track of the credits, where \( n = \) total number of input queues, an additional comparator, and queue mux. The credit manager operation takes place in 1 cycle, which is pipelined with the other stages of the AMC to reduce the overall latency. Algorithm 2 takes check_depth number of cycles for execution, and requires logic for division, a comparator, and registers to store the parameters check_depth and error_threshold. The NI connected to an AMC is often fully occupied at run-time, resulting in a wait time for data at AMCs in most cases, before they are sent to the NI. This wait time helps to mask the overhead involved in data approximation. A packet is only sent to the NI after it passes the data approximation stage. We consider all of these performance and power overheads when modeling the AMC for our experiments (Section 5). Note also that the parameters used in AMC, such as check_depth and error_threshold, can be updated by GPGPU firmware according to the needs of the application.

4.2 Data Aware Approximate NoC (Dapper)

The request plane of Dapper is a conventional 2D mesh based NoC with XY routing as mentioned in Section 3.1. In the reply plane, Dapper establishes dedicated overlay circuits from AMCs to cores. Each AMC’s overlay circuit is a predefined mapping of input and output ports in the reply plane NoC routers, during which flits traverse from source (AMC) to destinations (cores) in 3 cycles or less. An overlay circuit assigned to an AMC lasts for a time duration determined at run-time. To establish overlay circuits, the reply plane NoC is equipped with (i) a global overlay controller that decides the time duration for which an overlay circuit is established, and (ii) modified routers called bypass routers through which flits traverse in X or Y axes in a single cycle, stopping only at a turn.

4.2.1 Global Overlay Controller (GOC)
The primary function of the global overlay controller (GOC) is to determine and assign time durations for overlay circuits, for each AMC. The execution time is divided into epochs, and each epoch is further divided into time windows which are computed at the beginning of every epoch by the GOC. A time window for an AMC is determined at run-time based on the number of reply packets waiting at the AMC output queues, and the reply arrival rate at the AMC from the previous epoch. Each AMC sends the stats collected during an epoch to the GOC at the end of that epoch, using the overlay circuits. The GOC uses this received information to compute a weight function as shown in equation (1):

\[
\xi(m) = \alpha A(m) + \gamma B(m),
\]

where \( A(m) \) is the reply arrival rate and \( B(m) \) is the average queue occupancy at the \( m \)th AMC in the previous epoch. \( \alpha \) and \( \gamma \) are coefficients of the weight function. The GOC compares \( \xi \)’s of each AMC and computes time window durations \( T_1, T_2, T_3, \ldots, T_m \) for the next epoch as:

\[
T_i = K \cdot \xi(i)/[\xi(1) + \xi(2) + \xi(3) + \ldots + \xi(m)],
\]

where \( T_i \) is the time window of the \( i \)th AMC overlay and \( \xi(i) \) is its weight function. The ratio of the weight functions is then multiplied by a constant \( K \) which is equal to the periodicity of the time windows in an epoch. The time windows repeat periodically for \( E/K \) iterations in an epoch, where \( E \) is the epoch interval duration and \( K \) is the periodicity of the time window set. By having the time windows repeat and overlay circuits switch multiple times in an epoch, AMCs send flits in multiple bursts across an epoch. The time window durations are broadcast by the GOC at the beginning of an epoch, which is saved by the reply plane routers in dedicated registers, and then used for the setup and tear down of circuits. At the end of each epoch, the arrival rate \( A(m) \) and buffer occupancy \( B(m) \) of each AMC are sent to GOC using the corresponding overlay circuits. This requires 2 additional cycles at the end of each epoch (which is 10000 cycles in duration).

4.2.2 Bypass Router Architecture
The reply plane NoC is made up of bypass routers that support flits passing through them without stopping at each
hop for arbitration or route computation. Fig. 8 shows an overview of a bypass router architecture. A bypass router has asynchronous bypass links connecting output ports to input ports and latches via a crossbar. Each input port is connected to a dedicated latch to save the flit that is coming over the X axis. The router also saves time windows of each overlay circuit received from the GOC in a local overlay controller as shown in Fig. 8. The crossbar is configured at the beginning of each time window and reset at the end of the window to establish a different overlay configuration. The crossbar configuration of a router is based on its location on the 2D NoC and is selected by the selection unit based on the current overlay circuit. The output ports of a bypass router are connected to either input ports or input latches to enable flit transmission. In the first cycle, a flit traverses the bypass routers along the X axis through asynchronous links and gets latched at the input latches. In the second cycle, when the output ports are connected to input latches, the flit traverses along the bypass routers in the Y direction and gets latched again at the input latches. In the third cycle, the flit is sent from an input latch to the local (core) port. Thus, flit transmission can take 1 to 3 cycles. Two key components of a bypass router are the overlay controller and selection unit. The overlay controller keeps track of the time windows and calls the selection unit to dynamically configure the crossbar to establish the overlay circuits. The crossbar dynamically makes connections between input latches to output ports, and turns flits from X axis to Y axis, and from Y axis to local output ports as explained in the subsequent sections.

Algorithm 3 gives an overview of the overlay controller operation. The controller gets a list of time window durations and epoch duration as inputs from the GOC. It sends the list of time windows to the selection unit as input (line 3). At every cycle, it increments a local counter to keep track of the epoch duration (lines 5-6). At the end of an epoch duration, updated time window values are received from the GOC which are sent to the selection unit, and the local counter is reset (lines 8-10). Algorithm 4 gives an overview of the selection unit operation. The inputs to the selection unit are the time window durations \( \{T_1, \ldots, T_k\} \) for \( k \) overlay circuits (corresponding to the \( k \) AMCs). In algorithm 4, \( \{N, E, W, S, L\} \) means North, East, West, South, and local port. Also, \( La \) is the latch. The selection unit possesses knowledge of the 2D coordinates of the AMC that utilizes the overlay circuit on the NoC along with its own coordinates. With the given inputs, the selection unit creates connections for an overlay circuit between input ports, input latches and output ports. At the beginning of each time window, the selection unit compares the coordinates of the current router and the AMC for which the overlay is being established. Based on the location of the router, there are 4 possible scenarios for each AMC as shown in Fig. 8. Scenario 1: If the router and AMC are at the same NoC coordinate, the local input port is connected to the east and west output ports and local latch (line 6). Scenario 2: If the router and the AMC are along the same X axis, the east and west inputs are connected to the west and east output ports, and to their corresponding latches (line 8). Scenario 3: If the router and AMC are not along the same X axis and the router’s Y coordinate is greater than the AMC’s Y coordinate, the north input is connected to the south output and north latch (line 10). Scenario 4: If the router and AMC are not along the same X axis and the router’s Y coordinate is less than the AMC’s Y coordinate, the south input is connected to the north output and south latch (line 12).

At the end of a time window, the selection unit implements connections for the overlay circuit of the next AMC and its corresponding time duration. This entire operation takes 2 cycles at the beginning of each epoch. Fig. 8 shows an example of such a configuration where green lines represent the configuration made by the selection unit to connect the east input to east latch and west output.

Algorithm 3. Overlay controller operation

<table>
<thead>
<tr>
<th>Inputs: {time windows}, epoch_duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: e_counter ( \leftarrow 0 ); // reset counter value</td>
</tr>
<tr>
<td>2: ( {T_1, \ldots, T_k} \leftarrow {\text{time windows}} ); // initialize time window durations</td>
</tr>
<tr>
<td>3: selection unit(( {T_1, \ldots, T_k} ))</td>
</tr>
<tr>
<td>4: for every cycle do</td>
</tr>
<tr>
<td>5: if ( e_\text{counter} &lt; \text{epoch_duration} ) then</td>
</tr>
<tr>
<td>6: ( e_\text{counter} + +; )</td>
</tr>
<tr>
<td>7: else if ( e_\text{counter} = \text{epoch_duration} ) then</td>
</tr>
<tr>
<td>8: ( {T_1, \ldots, T_k} \leftarrow \text{read_values(GOC_input)} ); // save time windows</td>
</tr>
<tr>
<td>9: selection unit(( {T_1, \ldots, T_k} ))</td>
</tr>
<tr>
<td>10: ( e_\text{counter} \leftarrow 0 ); // reset counters</td>
</tr>
<tr>
<td>11: end if</td>
</tr>
<tr>
<td>12: end for</td>
</tr>
</tbody>
</table>

Routing: For conflict free routing using bypass routers, each row in the 2D NoC should only have one AMC. A packet that is approximable might have more than one destination based on the check_depth parameter of the AMC. To accommodate more than one destination, the header flit of the packet has more than one destination fields, and a destination length field. When a packet is ready to be transmitted at the NI of an MC, it is sent to the NI interface buffer based on the availability of the buffer space. At each cycle, the router transmits flits via the asynchronous links along the X direction and saves them at the corresponding input latches of each bypass router (shown via the green line in Fig. 8). The Y-compare unit (Fig. 8) then compares the destination Y coordinates of the flits with its own coordinates and sends a signal to the crossbar to establish hinge connections between
input latches, and the north/south/local output ports based on the Y coordinates of the current router at which the flit is latched and the destination router. Once the hinge connections between latches and output ports are established, the flits traverse in the Y direction (shown via the red line in Fig. 8) and reach the destination router where they are sent to the local output port. When the tail flit passes the bypass router, the hinge connections are reset to tear down the connections between latches and output ports.

Algorithm 4. Selection unit operation

Inputs: \( \{T_1 \ldots T_k\}, \{MC_1 \ldots MC_k\} \)
1: \( t \leftarrow 0, i \leftarrow 1 \)
2: for each cycle do
3: \( \text{if } t = = T_i \text{ then} \)
4: \( i \leftarrow (i + 1) \mod k \)
5: \( \text{if } local.(X, Y) == MC_i.(X, Y) \) then // scenario 1
6: \( L_{in} \rightarrow E_{out}, W_{out}, Lo(L) \)
7: \( \text{else if } local.(Y) == MC_i.(Y) \) then // scenario 2
8: \( E_{in} \rightarrow W_{out}, Lo(E) \) and \( W_{in} \rightarrow E_{out}, Lo(W) \)
9: \( \text{else if } local.(Y) > MC_i.(Y) \) then // scenario 3
10: \( N_{in} \rightarrow S_{out}, Lo(N) \)
11: \( \text{else if } local.(Y) < MC_i.(Y) \) then // scenario 4
12: \( S_{in} \rightarrow N_{out}, Lo(S) \)
13: \( \text{end if} \)
14: \( \text{else} \)
15: \( \text{end if} \)
16: \( t ++ \)
17: \( \text{end for} \)

Flow control: Since Dapper has lower latency than traditional NoCs, it fills up the receiving cores' queues much more rapidly than a traditional, 4-stage hop-by-hop, router. Hence, it is crucial that a robust flow control mechanism is integrated to ensure that the receiving queues of cores do not fill up quickly and start creating backpressure. Unlike the wormhole switching scheme where the intermediate routers at each hop save flits for re-transmission, in Dapper the flits are only saved at the receiving core. As a result, if there is any data loss on the asynchronous links, they may be undetected at the receiving core. Hence, there is also a need for a mechanism to ensure lossless transmission of packets traversing on overlay circuits.

To meet these goals, we use an acknowledgement-based flow control mechanism where the receiving cores send ACK signals back to the AMC when the packet is completely received. We establish ACK circuit using dedicated links of 7 bits width to send ACK signal back to AMCs. Bypass routers establish an ACK circuit using ACK links for each overlay circuit. These ACK circuits are not shared with data packets. A core does not send an ACK signal when its input queues are already full or when there is data loss in the packet. To facilitate fast transmission of the ACK signal from the cores to an AMC, each bypass router has an ACK in and ACK out port (Fig. 8), which are connected using asynchronous links, switch, and a selection-unit similar to the fast overlay circuit to relay the ACK message back to the AMC from the cores in 1-3 cycles. The AMC, upon receiving the ACK signals from the destination cores releases its output queue for the read reply data coming from DRAM. If the AMC does not receive an ACK signal, it services the next waiting packet in the output queue out-of-order. Based on the request type (read/write) the addresses of the missed request and the subsequent request are compared before the next packet is selected for transmission into the NoC to avoid read-after-write (RAW) or write-after-read (WAR) errors. Further, to avoid starvation of long waiting packets, the output queues in the AMC are designed as cyclic buffers so that all the waiting packets are processed in a round robin manner. Thus, this flow control mechanism ensures that packets at the head of the queue at the AMC do not block the subsequent packets that can use the available overlay circuit. To facilitate data loss detection, we assume the presence of a simple XOR-based parity bit error detection mechanism.

4.2.3 Overheads

The overheads involved with implementing the overlay controller, selection unit, and Y-compare are minimal. The overlay controller uses a counter, and a register to store the time window values received from the GOC, while the selection unit uses a counter, a cyclic register, and five 3-bit comparator circuits. The bypass router also has an additional Y-comparator, and logic to establish hinge connections. All of these components together take up a very small fraction of the power and area of a router. The bypass routers also do not have input buffers to save incoming flits, VCA, SA, and route computation logic. Hence, a bypass router consumes up to 50 percent less power compared to a traditional router. However, we increase the output buffer capacity of AMC by 50 percent as it takes at least 6 cycles for the complete read-reply transaction. All of these performance and power overheads are considered in our experimental analysis, presented next.

5 Experiments

5.1 Experimental Setup

We target a baseline 16-core GPGPU based accelerator to test the performance, energy consumption, network latency, and output error of Dapper + AMC compared to the state-of-the-art. We also test the scalability of our proposed architectures on a 64-core GPGPU. Table 1 lists the platform configurations. We used GPGPU-Sim [21] to collect detailed application traces and simulated the network and memory traffic on a customized Noxim NoC simulator [22] that integrates our Dapper + AMC architecture model. We obtained traces for 9 CUDA-based applications [1]. Table 2 gives an overview of each application and the approximable regions of their working data sets.

As mentioned earlier, we use the programming paradigm proposed in EnerJ [19] to specify the variables in these applications that are potentially approximable. We set the epoch duration in Dapper + AMC as 10,000 cycles. We performed experimental sensitivity analysis for parameter values (not included for brevity) and accordingly set the values of \( \alpha, \gamma \) coefficients of the weight function from equation (1) to 0.6, 0.4, and \( K = 1000 \) in equation (2) as they give the best NoC throughput for the applications we considered, in both 16 and 64 core platforms. We have also considered that the Threshold (line 4) in Algorithm 1 is equal to half of the check_depth parameter value because having a threshold larger than check_depth/2 leads to poor approximability as the
request scheduler prioritizes BLP over RBL even when half of the requests are approximable. Conversely, a smaller threshold might favor RBL over BLP even when the ratio of packets that could be coalesced is lower which adversely affects the DRAM utilization. The power, performance, and area values for our NoC architecture, modified MCs, and cores at the 22nm node are obtained using the open source tools DSENT [23] and GPUWATTCH [24], and gate-level analysis.

We compare our proposed Dapper + AMC architecture with several alternatives, as shown in Table 3. The baseline NoC has a channel width of 128-bits which is the size of the L1 cache line, to provide high throughput to MCs. We also compare with the prior work that utilizes dictionary-based approximate compression and decompression at each network interface, called Approx-NoC [16]. Approx-NoC uses dictionary-based inexact compression using approximation at the sending network interface to reduce the number of flits that are injected into the NoC and save the overall energy consumed. We additionally compare our work with the memory-aware-circuit-overlay-NoC, MACRO [31]. MACRO has a fast overlay circuit network in the reply plane from MCs to cores similar to that of the current work but lacks packet broadcast capability, as well as fast broadcast ACK system for fault tolerance, and the data approximator in its MC. The Baseline NoC, Approx-NoC, and MACRO have first row first come first serve (FR-FCFS) memory scheduling scheme in the request channel of their MCs. To analyze the drawbacks of MC scheduling without NoC latency optimization, we compare our work with staged memory scheduling (SMS) [29] that proposes an application aware bank level parallelism to minimize the wait time of packets in the MC input queue. However, [29] does not have a latency optimized NoC like Dapper + AMC. Table 3 summarizes the NoC and MC configuration of all five comparison works. The size of the AMC output buffer is set to accommodate 66 data packets, as used in GPGPU-Sim, for all the comparison works. We also consider 4 parallel input queues in the AMC as the average bank level parallelism of the requests is around 3.5 as shown in Fig. 4b.

### 5.2 Sensitivity Analysis

We first conduct experiments to determine the best values of buffer check_depth and error_threshold parameters used in our data approximation stage in the AMCs in Algorithm 1 and Algorithm 2. We compare Dapper + AMC with different check_depths and error_thresholds to analyze the tradeoffs between NoC throughput improvements and the output error observed at the end of the application execution. Output error is computed using equations (3) and (4):

$$e = \frac{(V - V')}{V} \quad (3)$$

$$error = \frac{1}{M} \sum_{i=0}^{M} |e_i|, \quad (4)$$

where for each point in the output that comprises of images or matrices, $V$ and $V'$ are the actual and inexact points and $M$ is the total number of output points. Equation (4) gives the value of output error obtained at the end of the simulation using approximation.

Fig. 9a shows the plots of normalized NoC throughput and normalized output error observed with Dapper + AMC across different values of output buffer check_depths (4, 8, 12). The bars represent normalized average NoC throughput observed, and dots represent the normalized output error across different benchmark applications. Check_depth is defined to be the maximum number of input queue entries that are processed by the approximate-data aware scheduler of AMC in the RBL scheme. Check_depth is also defined as the maximum number of reply data units that can be coalesced in
the reply channel of AMC. In this manner, the approximation knob in the request and the reply channels are controlled by a single parameter. All the results are normalized to the result for the check_depth = 4 configuration of Dapper + AMC.

From Fig. 9a, on average, a check_depth of 12 gives up to 3.5 percent higher throughput in Dapper + AMC with minimum increase in output error compared to check_depths of 4 and 8. Dapper + AMC leverages the check_depth parameter for opportunistic burst of approximable RBL requests that increases the number of coalescable packets. The benefits of increased check_depth can be seen better in applications that are both memory intensive and contain higher ratio of approximable input data (e.g., ConvTex, Hist). Dapper + AMC also shows throughput improvement at higher check_depths in applications with lower ratio of approximable data (BlackScholes, FWT, MergeSort) due to its intelligent memory scheduling. For the applications DCT, DXTC, RAY and NN, NoC throughput does not increase rapidly with higher check_depth, and the output error also remains constant across different check_depths. This is because these applications are either compute intensive or execute in a manner that does not fully congest the NoC. Hence, for the rest of our experimental analysis, we use check_depth = 12. If the AMC has less than 12 buffers filled at any point, it uses the maximum buffer depth available. Also, check_depth > 12 leads to a higher output error in applications such as FWT and DXTC along with additional processing time latency in AMC for both request scheduling and data coalescing which adds to the AMC bottleneck.

Fig. 9b shows the plots of normalized average throughput and output error values observed Dapper + AMC across different error_thresholds (5, 10, 15, and 20 percent). All results are normalized to the result for the 5 percent error_threshold configuration. The applications that are compute intensive and highly approximable, such as DCT, DXTC, NN, and RAY, show a slight increase in NoC throughput at higher error_thresholds. However, for DCT, and RAY the increase in output error is high at higher error_thresholds due to the high volume of approximable variables present in the memory reply data. For memory intensive approximable applications such as ConvTex and Hist, NoC throughput increases by around 10 percent if the error_threshold is increased from 10 to 15 percent. For memory intensive applications that do not have highly approximable data such as BlackScholes, FWT, and MergeSort, the NoC throughput remains the same even with increase in error_threshold. On an average an error_threshold of 10 percent leads to ~15 percent increase in output error and ~9 percent increase in throughput compared to an error_threshold of 5 percent. For error_thresholds of 15 and 20 percent the output error observed is 2.3× and 2.7× higher compared to error_threshold of 5 percent. This eventually leads to poor application output quality. We have also observed that on average, an error_threshold of 5 percent gives no performance gains compared to an error-free execution. For this reason, for the rest of our experimental analysis we use the error_threshold of 10 percent with Dapper + AMC when comparing with other architectures as it gives an acceptable application output error with better throughput than error_threshold of 5 percent.

5.3 NoC Throughput Analysis

Fig. 10a shows the average NoC throughput observed across different NoC and MC architectures in a 16-core accelerator. On average Dapper + AMC shows 21 percent improvement compared to baseline, and 21.5 percent improvement compared to base + SMS. Although base + SMS tries to maximize the number of RBL and BLP hits by forming memory request batches, it does not dynamically modify the RBL and BLP intensity of the commands like our scheduling scheme does. Secondly, it does not have a fast overlay NoC. Dapper + AMC outperforms Approx-NoC that uses dictionary-based encoding and decoding at NIs by 68 percent, as the encoding and decoding process takes an additional 2-3
cycles at both the sending and receiving nodes of a NoC for all packets. Dapper + AMC outperforms MACRO by around 7 percent. These benefits are due to the joint optimization of request scheduling and reply packet coalescing in AMC that is absent in MACRO. For memory intensive and high approximable workloads such as ConvTex, DCT, and HIST the throughput improvement of Dapper + AMC reaches up to 40 percent compared to the baseline. For minimal approximable benchmarks like MergeSort, the throughput benefits of Dapper + AMC are diminished, however it achieves same throughput as the baseline. Fig. 10b shows that Dapper + AMC is highly scalable with average throughput improvement of up to 15.7 percent compared to the baseline and up to 15.34 percent compared to base + SMS. The improvements are slightly diminished in Dapper + AMC in the 64-core platform compared to the 16-core platform due to the higher number of cores sending more requests at the MC which increases DRAM latency for memory requests that worsens the MC bottleneck issue. Unlike Dapper + AMC, Approx-NoC loses up to 50 percent of its throughput in this configuration. Dapper + AMC has up to 5.35 percent better throughput than MACRO. This shows the importance of approximate data-aware request scheduling and data coalescing with the AMC.

5.4 NoC Latency Analysis

Fig. 11a shows the average NoC latency observed from the source to destination in a 16-core accelerator. Dapper + AMC is much faster than baseline and base + SMS by up to 45.5 and 46 percent, respectively. This is one of the key contributors of increase in throughput. The AMC’s approximate data-aware scheduling scheme is responsible for Dapper + AMC’s 25 percent improvement compared to MACRO due to the increased approximable and coalesced read reply data that reduces the MC bottleneck and NoC congestion. Also, in Dapper + AMC, the ACK based flow-control mechanism together with the cyclic output buffer ensures that MCs utilize their share of overlay circuits efficiently. Approx-NoC has a higher latency compared to the baseline due to its slower encoding and decoding steps at the NIs.

Fig. 11b shows the comparison of NoC latency in 64-core accelerators. The baseline NoC latency in 64-core platforms is up to 3× higher than the 16-core platforms because of the increased NoC traffic due to the higher number of cores in the platform. Dapper + AMC shows up to 78.7 percent improvement compared to the baseline, and 85 percent improvement compared to base + SMS. The NoC latency in base + SMS is higher than baseline due to the waiting time (at the incoming buffers of the MC) needed to create batches of requests according to the SMS scheduler. This impact is higher in a 64-core platform than a 16-core platform. In Approx-NoC, the latency degradation is reduced from 50 percent to around 22 percent compared to the baseline. As the ratio of MCs to cores decreases from 16-cores to 64-cores, the overhead of encoding/decoding is hidden in the packet traversal latency, allowing the approach to recover some of its lost latency. Dapper + AMC performs better than MACRO by 4.8 percent in 64-core platform which is better than improvements of ~3 percent in 16-core platform due to the higher ratio of approximable packets in Dapper + AMC in a 64-core platform.

5.5 Application Execution Time Analysis

Fig. 12a shows the application execution times observed in 16-core GPGPU accelerators for the various architectures. The throughput and latency improvements observed in the previous sections are translated into shorter application execution time for Dapper + AMC compared to baseline, and base + SMS. In 16-core platforms, Dapper + AMC achieves around 9.5 percent improvement compared to both baseline...
and base + SMS. The improvements for applications such as DXTc, and MergeSort the application execution time are not proportional to the NoC throughput and latency. These applications are also compute intensive where they have enough kernels to execute while waiting for data. The faster NoC latency can reduce the number of kernels concurrently scheduled on the cores that could potentially increase the execution time in these applications. In these applications, Approx-NoC performs similar to Dapper + AMC. The same trends are observed in the 64-core platform as shown in Fig. 12b. On average Dapper + AMC shows 5.4 percent improvement compared to baseline, and 4.5 percent improvement compared to base + SMS. In the MergeSort application, base + SMS gives 35 percent improvement compared to baseline in a 64-core platform due to the batch-based memory scheduling that is optimized for highly data parallel applications. However, most of the contemporary applications that are highly approximable perform better with Dapper + AMC.

5.6 Energy Consumption Analysis

Fig. 13a shows the energy consumption of Dapper + AMC compared to the prior works across different benchmarks in a 16-core platform. On average, Dapper + AMC consumes up to 38.3 percent less energy compared to the baseline and 38.1 percent less energy compared to base + SMS architectures. MACRO and Dapper + AMC consume lower energy due to their low power router architectures in the reply plane. Even though Dapper + AMC has energy overhead involved in the AMC architecture, AMCs are fewer in number compared to NoC routers and NIs. Also, the overall reduction in the number of packets injected from the AMC into the NoC reduces the total energy consumption of Dapper + AMC compared to the baseline. Approx-NoC on the other hand consumes higher energy due to the higher power dissipated in its routers that need additional logic and content addressable storage (CAS) based registers for saving the most used values for compression and decompression. In Dapper + AMC, energy savings are observed for all the applications including those that have lower to medium ratio of approximable data such as MergeSort, FWT, and Blacksholes, due to the low power NoC used in the reply plane and faster execution times. However, the energy savings are higher when applications have a higher ratio of approximable packets, such as in ConvTex, DXTc and DCT.

Fig. 13b shows the energy consumption of different architectures for a 64-core platform. In the 64-core platform, the energy savings follow the same trend as in the 16-core platform. On average, Dapper + AMC consumes up to 30 percent lower energy compared to the baseline, and 27.5 percent lower energy compared to base + SMS. However, Dapper + AMC consumes slightly higher energy while scheduling requests and coalescing reply packets than MACRO by around 1.5 percent. But this is not a major drawback given the improvements in NoC throughput and latency given by the addition of AMC in Dapper + AMC.

5.7 Output Error Percentage Analysis

Obviously, it is important to analyze the error in applications that are subjected to approximation. Fig. 14a shows the comparison of output error percent for Approx-NoC at 10 percent error_threshold and Dapper + AMC at 10 and 15 percent error_threshold. Note that the baseline, base + SMS, and MACRO NoC are not shown, as they have no output error. The error percentage is computed as \( \frac{100}{C} \times \text{error} \) from equation (4). Approx-NoC shows high percentage of error in its output due to a flaw (right shift division) in its error_threshold computation which is used for identifying the
approximable data before compression. This results in a very high error percentage when data values are incorrectly marked for approximation. Dapper + AMC on the other hand incurs 1 to 4 percent output error in the application. On average, Dapper + AMC gives around 2.3 percent error in application output in the 16-core platform. The same trend is shown in Fig. 14b for a 64-core platform. Although it seems intuitive that a 64-core platform may coalesce more packets than a 16-core platform, the ratio of approximable packets is dependent on the application’s input and not the platform size. Hence, the output error percent has no discernable change as we move from a 16-core to a 64-core platform. Thus, Dapper+AMC represents a promising NoC-and-MC-centric solution to improve performance and save energy consumption in GPGPUs for applications that have a potential for being approximated (i.e., applications that can tolerate some output error).

As an illustrative example, Fig. 15 shows the DCT application output under no error (when using the baseline NoC), and when using the Dapper + AMC in a 16-core platform. It can be observed that the output for the configuration of Dapper + AMC with 10 percent error_threshold is virtually indistinguishable from the no error case, while saving almost 38 percent energy (Fig. 13a) compared to the baseline NoC. This example highlights the exciting potential of Dapper+AMC to save energy in GPGPUs while increasing overall performance.

6 Conclusion
In this paper, we propose a novel data-aware approximate NoC Dapper, and an approximate memory controller, for GPGPU accelerators. Our Dapper + AMC architecture uses intelligent memory scheduling in the MC to increase throughput and also identifies the approximable data waiting in the output buffers of the MC and coalesces them to reduce the number of packets injected into the NoC. Also, we advocate for a fast overlay circuit based NoC architecture in the reply plane of the NoC for the reply packets to reach their destinations in 3 cycles or less. Experimental results show that on average Dapper + AMC increase the NoC throughput by 21 percent in 16-core platforms and up to 7 percent in 64-core platforms while consuming up to 38 percent less energy compared to baseline NoC, with up to 9.5 percent lower application execution time and around 2.3 percent error in the application output compared to the baseline. Thus, for the class of emerging applications that have the ability to tolerate a small amount of error in their outputs, Dapper + AMC can provide significant savings at the NoC and MC level. These savings are orthogonal to (and can be combined with) further approximation strategies at the computation components (e.g., using approximate adders) to further expand the design space of trade-offs between application output error, energy costs, and execution time.

Acknowledgments
This research is supported by Grant from NSF (CCF-1813370).

References


For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.