2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-7652-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/HPCA56546.2023.10071089

Dalorex: A Data-Local Program Execution and Architecture for Memory-bound Applications

Marcelo Orenes-Vera, Esin Tureci, David Wentzlaff and Margaret Martonosi
Princeton University
Princeton, New Jersey, USA
Email: {movera, esin.tureci, wentzlaf, mrm}@princeton.edu

***Abstract*—Applications with low data reuse and frequent irregular memory accesses, such as graph or sparse linear algebra workloads, fail to scale well due to memory bottlenecks and poor core utilization. While prior work with prefetching, decoupling, or pipelining can mitigate memory latency and improve core utilization, memory bottlenecks persist due to limited off-chip bandwidth. Approaches doing processing in-memory (PIM) with Hybrid Memory Cube (HMC) overcome bandwidth limitations but fail to achieve high core utilization due to poor task scheduling and synchronization overheads. Moreover, the high memory-per-core ratio available with HMC limits strong scaling.**

We introduce Dalorex, a hardware-software co-design that achieves high parallelism and energy efficiency, demonstrating strong scaling with >16,000 cores when processing graph and sparse linear algebra workloads. Over the prior work in PIM, both using 256 cores, Dalorex improves performance and energy consumption by two orders of magnitude through (1) a tile-based distributed-memory architecture where each processing tile holds an equal amount of data, and all memory operations are local; (2) a task-based parallel programming model where tasks are executed by the processing unit that is co-located with the target data; (3) a network design optimized for irregular traffic, where all communication is one-way, and messages do not contain routing metadata; (4) novel traffic-aware task scheduling hardware that maintains high core utilization; and (5) a data-placement strategy that improves work balance.

This work proposes architectural and software innovations to provide the greatest scalability to date for running graph algorithms while still being programmable for other domains.

***Index Terms*—Distributed, scalable, parallel, data-local, near-memory, architecture, sparse, graph, NoC, network, bandwidth.**

I. Introduction

System designs are increasingly exploiting heterogeneous, accelerator-rich designs to scale performance due to the slowing of Moore's Law [42] and the end of Dennard Scaling [21], [54]. While compute-bound workloads thrive in an accelerator-rich environment, memory-bound workloads, such as graph algorithms and sparse linear algebra, present the following **challenges**: (a) a low compute-per-data ratio, with little and unpredictable data reuse; (b) frequent fine-grained irregular memory accesses that make memory hierarchies inefficient; (c) atomic accesses, synchronization, and inherent load imbalance resulting in poor utilization of computing resources.

Recent work proposed using accelerators that pipeline the different stages of graph processing in hardware [1], [14], [19], [45], [49], [53]. Other works have used general-purpose cores with decoupling and prefetching to mitigate memory latency [18], [39], [44], [47], [59], and coalescing to reduce

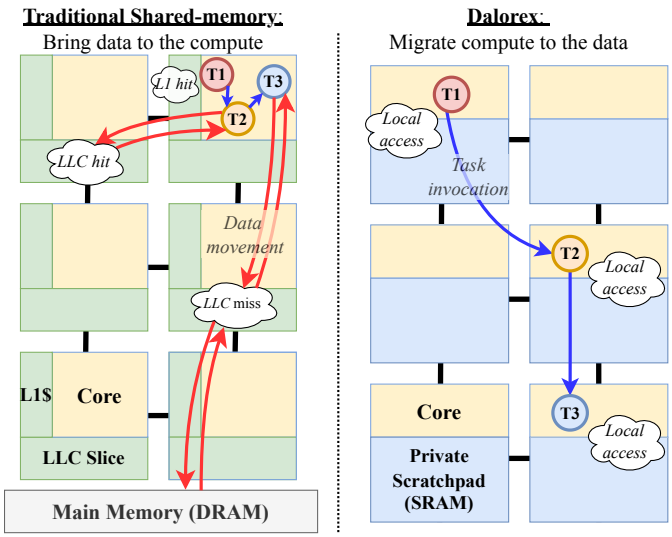


Fig. 1. Program execution of three sequential graph-processing steps in a cache hierarchy (left), and Dalorex (right). Instead of moving data with little reuse, Dalorex invokes tasks where the data is local—reducing movement.

atomic update serialization [43], but failed to avoid costly data movements and are eventually bottlenecked by memory bandwidth. The processing-in-memory (PIM) proposals enjoy a higher memory bandwidth by processing data near DRAM [2], [71], [76]. However, their memory integration constrains the **storage-per-core** ratio, limiting the level of parallelism—as we demonstrate in this paper. In addition, their parallelization suffers from load imbalance and synchronization overheads.

Opportunities: We started our work by examining what features are necessary to execute graph workloads in a scalable manner. We observed that to maximize throughput (edges processed per second), *the solution should*: (1) minimize data movement, which is the performance bottleneck and the most significant energy cost; (2) exploit the spatio-temporal parallelism of operations to improve work balance and resource utilization; (3) avoid serializing features such as global synchronization barriers and read-modify-write atomic operations; and (4) avoid frontier redundancy and data staleness to minimize the number of edges explored (work efficiency).

Our Approach: We present Dalorex, a highly scalable and energy-efficient hardware-software co-design that exploits these opportunities by optimizing across the computing stack without sacrificing ISA programmability. Our architecture is a 2D array of homogeneous tiles, each containing an SRAM

scratchpad, a thin in-order core (with no cache) called Processing Unit (PU), a task scheduling unit feeding the PU, and a router connected to the network-on-chip (NoC). Our programming model enables further parallelism by splitting the sequential code inside a parallel-loop iteration into tasks at each pointer indirection (Fig. 2). Tasks execute at the tile containing the data to be operated on. Correct program order is guaranteed because new tasks are spawned by the parent task by placing their invocation parameters into the NoC. Since the spawned tasks are independent and execute in any order, Dalorex achieves huge synchronization-free spatio-temporal parallelism. We designed the task scheduling unit (TSU) with a closed-loop feedback system to achieve work efficiency and high utilization, as this varies with task flow order in this decentralized model. TSU also coordinates PUs and routers to make task invocations non-blocking and non-interrupting.

Dalorex distributes all data arrays in equal chunks across tiles using low-order index bits. In this way, Dalorex achieves both good workload and NoC traffic balance, despite the irregular access patterns of sparse workloads. As a result of this data partitioning, each tile owns a set of nodes (with no copies of data), and all operations are atomic by design. Because of this, Dalorex can also have a decentralized frontier, such that one can remove the global barrier between epochs of graph algorithms. In addition, the data-local execution of tasks removes the round-trip time and energy of accessing memory, as well as the serialization due to access contention.

Fig. 1 illustrates how the distributed memory architecture of Dalorex (right panel) minimizes data movement compared to architectures with hierarchical shared-memory structures (left panel). In Dalorex, each piece of data is only accessed by one PU. Instead of bringing data to the PUs, they send task-invoking messages to the tile containing the data to be processed next. Dalorex exploits the inherent pointer indirection in sparse data formats to route a task-invocation message. The tile's PU executes the task corresponding to the message received. In architectures with a traditional memory hierarchy, sparse applications result in overwhelming data movement: data-reuse distance is highly irregular, and thus, cache thrashing leads to more than 50% of all memory accesses missing in the cache levels and going to main memory [39]. Shared-memory architectures also carry the overhead of cache coherence, virtual memory, and atomic operations.

The technical contributions of this paper are:

- A data-local execution model where each processing tile holds an equal amount of data—operating on local data makes all updates atomic and minimizes data movement.
- A programming model that unleashes parallelism and improves work balance by splitting the code into tasks—based on data location—while preserving program order.
- A tiled distributed-memory architecture connected by a NoC optimized for irregular communication (headerless task routing based on the index of the array they access).
- A hardware unit (TSU) that removes task-invocation overheads and schedules tasks to maximize core utilization and work efficiency by sensing the network traffic.

```
1  while not frontier.isEmpty()
2    parallel for (v : frontier)
3  T1  node_dist = dist[v]
4      neigh_begin, neigh_end = ptr[v], ptr[v+1]
5      for i in range(neigh_begin, neigh_end):
6  T2    neigh_id = edge_idx[i]
7        new_dist = node_dist + edge_values[i]
8  T3    curr_dist = dist[neigh_id]
9        if (new_dist < curr_dist):
10         dist[neigh_id] = new_dist
11         new_frontier.push(neigh_id)
12 frontier = new_frontier
13 new_frontier = []
```

Fig. 2. Pseudocode of the Single Source Shortest Path (SSSP) algorithm, and how it is split into Dalorex tasks based on indirect memory accesses. The colored variables determine the tile at which the next task is executed.

We evaluate Dalorex and demonstrate that:

- Our architecture and data layout improve *performance* by 6.2× over the best ISA-programmable prior work [2]. On top of that, our task invocation scheme improves by 4.7×, and our uniform data placement and traffic-aware scheduling, 4.4× more. Finally, removing the barriers and upgrading the NoC provides an extra 1.8×, compounding a 221× geomean improvement across four key graph applications, using equal processor count (256).
- Regarding *energy*, the improvements of using SRAM (16×), our parallelization and data placement (5.2×), and TSU (3.9×) compound 325×, in geomean.
- Dalorex achieves strong scaling with over 16,000 processing tiles, not being limited by memory bandwidth.
- Our data distribution and parallelization do not require pre-processing to improve work balance across tiles; Dalorex remains agnostic to the dataset characteristics.

II. Background and Motivation

Dalorex is designed to accelerate applications that are memory-bound due to irregular access patterns caused by pointer indirection. Although this paper focuses on graph algorithms, Dalorex is applicable to other domains such as sparse linear algebra. We demonstrate this by evaluating sparse matrix-vector multiplication.

A. Graph Algorithms and their Memory Access Patterns

Graphs are represented using adjacency matrices where rows/columns represent vertices and values, weighted edges. Since columns contain mostly zero values (most vertices have few connections), these matrices are stored in formats like Compressed-Sparse-Row (CSR) using four arrays. The non-zero elements are accessed via pointer indirection.

Fig. 2 shows the sequential code for Single Source Shortest Path (SSSP) and colors the code sections that would be split into Dalorex tasks. In a regular memory hierarchy, the accesses to neighbor vertex data in the innermost loop (`line 8`) result in many cache misses and thus, costly accesses to DRAM [39]. Moreover, the source vertex data (`lines 3 and 4`) and the neighbor index (`line 6`) are also accessed indirectly, and the utility of the cache depends on the number of neighbors

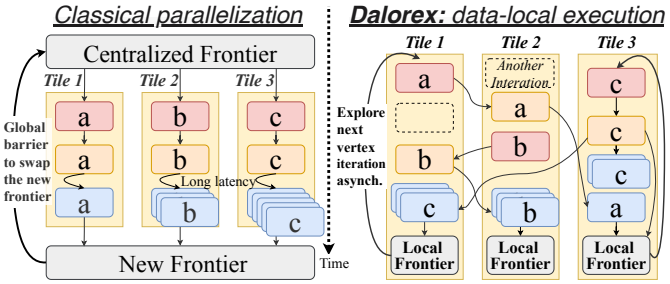


Fig. 3. Program order and synchronization for Bulk Synchronous Parallel (BSP), left, versus Dalorex. BSP parallelization leads to work imbalance as Task2 (orange) may generate several Task3 (blue). Columns show the tasks that are executed in each tile (arrows depict program order). Colors indicate task type based on the code of Fig. 2. Letters represent different vertex iterations. Dotted boxes depict that tasks from other iterations may interleave.

and the workload distribution. In Dalorex, the code is split at each level of pointer indirection, leading to a series of tasks. For example, `T1` accesses arrays **dist** and **ptr** (tuple of size #*vertices*), while `T2` accesses arrays **edge_idx** and **edge_values** (tuple of size #*edges*), and `T3` accesses **dist**.

As Fig. 3 illustrates, Dalorex allows *spatial interleavings within vertex iterations* while preserving program order through sequential invocation of the tasks. The classical timewise interleavings of parallel-loop iterations are also allowed. Finally, instead of using a centralized frontier, Dalorex uses local frontiers, which removes the synchronization overheads of frontier insertions and global barriers. As a result, our novel programming model maximizes program parallelization.

B. Algorithmic variants

There are two modes of processing graph data: pulling data for a vertex from its neighbors or pushing data to its neighbors [7]. While pull-based algorithms tend to require higher memory communication, push-based algorithms often require atomic operations. When atomic operations are mitigated, push-based algorithms have the advantage of reduced communication overhead. A hybrid version, direction-optimized *BFS* [5], and its variants can offer faster convergence. However, they incur a storage overhead and need heuristics, as they may access either the source or the destination of an edge. Although pull-based algorithms can be executed on the Dalorex architecture, we focus on push-based algorithms due to their reduced communication and work efficiency since Dalorex eliminates the need for atomic operations.

C. Prior Work

We describe pipelining and data-movement-reducing techniques and lay out what aspects limit their scalability.

1) Hardware Techniques: Recent works have used decoupling to overlap data fetch and computation by running ahead in the loop iterations to bring the data asynchronously [39], [44], [47], [59]. To accomplish this, they perform program slicing on each software thread, creating a software pipeline effect. Others have proposed accelerators to perform the graph search as a hardware pipeline [1], [19], [45], [49], [53]. Polygraph [14] generalized prior accelerator designs to perform any of their algorithmic variants and optimize work efficiency based on dataset characteristics, and Fifer [45] offers a dynamic temporal pipelining to achieve load-balancing. While effective for hiding latency and increasing load-balancing, these approaches remain highly energy inefficient due to excessive data movement and are ultimately limited by DRAM bandwidth.

To increase memory bandwidth and reduce data movement, Tesseract [2] proposes in-memory graph processing by introducing cores into the logic layer of a 3D Hybrid Memory Cube (HMC) [24], [50]. Tesseract executes remote calls at the cores located near the data, similar to the execution-migration literature [37], [56]. However, their performance is limited because: (a) Their vertex-based data distribution causes load imbalance since the highly-variable vertex degree in graphs causes a different workload per core; (b) Tesseract remote calls are interrupting, incurring 50-cycle penalties, and GraphQ's solution to overcoming this employs barriers for batch communication [76], causing high synchronization overheads; (c) HMC-based architectures are constrained in the number of cores per cube (a core per vault). In this work, as we explore the limits of graph parallelization, we demonstrate that the energy-optimal storage size per tile is in the single-digit megabyte range, much smaller than what HMC offers.

2) Software Techniques: Software solutions to accelerating graph applications include optimizing data placement, work efficiency, and parallelization schemes based on the target hardware system [72]. For distributed systems, GiraphUC [20] puts forward a barrierless model that reduces message staleness and removes global synchronization. However, GiraphUC is not optimized for data locality and has high communication costs. Pregel [38] does parallelize tasks such that each iteration is executed where the data is local, but the data is distributed in a vertex-centric manner, resulting in inherent load-balancing problems and communication overheads.

D. Manycore Architectures & Memory Bandwidth

Large-scale parallel processing can use many small units. From Systolic arrays [25], [26], [33] and streaming architectures [23], [60], [61], to modern manycores [4], [15], [17], [69], supplying irregularly-accessed data remains challenging.

Recently, some industry products have utilized large amounts of SRAM to achieve high on-chip bandwidth, and so, higher performance [9], [30]. Aside from energy consumption, there are architectural advantages to using a scratchpad memory per core, e.g., low latency, dedicated access, and scalable memory bandwidth (more cores means more memory ports).

III. Dalorex: A Hardware-Software Co-design

Dalorex minimizes data movement and maximizes resource utilization with: (1) a data distribution that allows for *only* local memory operations; (2) a programming model that allows for intra-loop spatial and inter-loop timewise interleavings; (3) a homogeneous tile-based architecture that optimizes irregular data-access patterns, where each tile includes a local memory, a router, and a processing unit. Program execution is orchestrated by the task scheduling unit (TSU).

A. Data distribution

Graphs and sparse matrices are often stored in formats like CSR using four data arrays. In Dalorex, these arrays are divided equally across all tiles and stored in their private memories, making each tile responsible for operations only on its local data. For example, the `edge_values` array has as many elements as edges (E) in a graph. This array is split as in Listing 1 so that each of the T tiles has `EDGES_PER_CHUNK` (E/T) adjacent elements, e.g., the first tile contains elements from 0 to `EDGES_PER_CHUNK-1`.

Ours is the first work that distributes an adjacency matrix in this manner; the usual approach is to do a 2D distribution of the matrix [8], where each computing element gets a rectangular subset of the matrix to compute. Although designed to minimize communication, 2D distribution presents some challenges: a subset of a sparse matrix is hyper-sparse (making row or column sparse formats storage-inefficient), and the resulting chunks do not have an equal number of non-zeros (different storage needs). Because Dalorex has all memory equally distributed across tiles, it improves workload balance.

B. Programming model

The BSP model parallelizes graph algorithms either at the outer loop (processing vertices in the frontier) or the inner loop (processing the neighbors of the frontier vertices) since these can be processed in any order. We posit that if the iterations of the inner loop are performed in program order, the location of the execution can be altered. Dalorex preserves the instruction order within an iteration since subsequent tasks are invoked by the parent task at completion. Since only one tile has access to each data chunk, coherence is not an issue.

Adapting a graph kernel to Dalorex involves splitting the inner-loop iteration into multiple tasks at each pointer indirection. As we have seen in Fig. 2, this results in three tasks for SSSP, where each task produces the array index to be accessed by the next task. Additionally, graph algorithms require a fourth task to explore the frontier vertices (Listing 1). Every tile contains the same code and can perform any task.

From the program execution timeline perspective, after a tile performs a task, it sends the output of the task (i.e., the input for the next task) to the tile containing the data to be operated next, thus preserving sequential order.

From the point of view of an individual tile, inputs for any task may arrive in any order into their corresponding task-specific queues. The execution order of different tasks is determined by the TSU—described in Section III-E.

Application programmers would not program Dalorex directly. Instead, DSLs such as TACO [29] (sparse algebra) or GraphIt [72] (graph analytics) could invoke our kernel library.

C. Program Flow and Synchronization

Dalorex programs do not have a *main*. Instead, tiles await the task parameters to arrive in the corresponding input queue, and PUs are invoked by TSU to process them. A task invokes the next task by placing the task parameters into an output queue (OQ). An OQ can be either another task's input queue

```
## These constants are filled when the program is loaded
param NODES_PER_CHUNK
param EDGES_PER_CHUNK
## Local chunk of the dataset arrays
var dist[NODES_PER_CHUNK]
var ptr[NODES_PER_CHUNK]
var edge_idx[EDGES_PER_CHUNK]
var edge_values[EDGES_PER_CHUNK]

const FRONTIER_LEN = NODES_PER_CHUNK/32
const OQT2 = 1024
## Bitmap frontier and memory-stored variables
var frontier[FRONTIER_LEN] = [0,...,0]
var blocks_in_frontier = 0
var t1_new_vertex = True
var neighbor_begin = 0;

##Configure network channels between tasks and their queues
CQ1 = channel(q_len=128, target=T2, encode=EDGES_PER_CHUNK)
CQ2 = channel(q_len=OQT2,target=T3, encode=NODES_PER_CHUNK)

## Declaring a task requires the length of its IQ and
## whether its parameters are loaded before the invocation
task T1 [32] ():
   #T1 params aren't pre-loaded. We read from the IQ of T1
   vertex_id = peek(IQ1.head)
   if (t1_new_vertex):
       neighbor_begin = ptr[vertex_id]
   neighbor_end = ptr[vertex_id+1]
   while (!CQ1.full && (neighbor_begin < neighbor_end)):
     ##Split msg if range crosses chunk limits or > OQT2
      tile = (neighbor_begin/EDGES_PER_CHUNK) + 1;
      partial_end = min(neighbor_end, tile*EDGES_PER_CHUNK)
      partial_end = min(partial_end, neighbor_begin + OQT2)
      CQ1 = neighbor_begin ## global idx for tile address
      CQ1 = (partial_end % NODES_PER_CHUNK) ## local idx
      CQ1 = dist[vertex_id]
     neighbor_begin = partial_end
   ## We pop vertex_id if the whole range was pushed to CQ1
   t1_new_vertex = (neighbor_begin == partial_end)
   if (t1_new_vertex):
       pop(IQ1.head)

# Task parameters are loaded by TSU before the task begins
task T2 [128] (neighbor_begin, neighbor_end, vertex_dist):
    for i in range(neighbor_begin,neighbor_end):
    ##Writing to a Channel Queue sends data to the network
        CQ2 = edge_idx[i]
        CQ2 = edge_values[i] + vertex_dist

task T3 [2048] (neigh_id, new_dist):
    curr_dist = dist[neigh_id]
    if (new_dist < curr_dist):
        dist[neigh_id] = new_dist

      ## Insert vertex into Local Frontier
      blk_id = neigh_id >> 5;
        blk_bits = frontier[blk_id];
        frontier[blk_id] = mask_in_bit(blk_bits, neigh_id)
        if (blk_bits == 0): ##only add newly active blocks
            blocks_in_frontier++
            IQ4 = blk_id

# T4 re-explores the local frontier queue
task T4 [FRONTIER_LEN] ():
    frontier_block = peek(IQ4)
    while(blocks_in_frontier > 0 && !IQ1.full):
        blk_bits = frontier[frontier_block]
        block_base = frontier_block << 5;
        while(blk_bits > 0 && !IQ1.full):
            idx = search_msb(blk_bits)
            blk_bits = mask_out_bit(blk_bits, idx)
            vertex = block_base + idx
            IQ1 = vertex
        # If all the no pending vertices in the block we
        # remove the block from the frontier queue
        if (blk_bits == 0):
            pop(IQ4)
            blocks_in_frontier--
            frontier_block = peek(IQ4)
```

Listing 1. Pseudo-code of the SSSP algorithm adapted to use the Dalorex programming model. This programming model would be embedded in a high-level language via an API. The programmer or compiler would fill tasks' code.

(IQ) if it operates over data residing in the same tile or a channel queue (CQ), which puts a message into the network.

Listing 1 contains the code of the Dalorex-adapted SSSP kernel. To start running SSSP, only the tile containing the root of the graph search receives a message to invoke the first task. **T1** obtains the range array indices that contain the neighbors of `vertex_id`. If this range crosses the border of a chunk, a separate message is sent to each tile with the corresponding *begin* and *end* indices. Similarly, the range is split if the length is bigger than a constant `OQT2`, which is set to guarantee that `T2` can execute without exceeding the capacity of `CQ2`. However, `T1` does not have this guarantee and needs to check explicitly that `CQ1` does not overflow. If `CQ1` fills before sending all the messages for `vertex_id` range, a flag is set, and `T1` ends early after updating the begin index. The next time `T1` is invoked it continues operating on the same `vertex_id` since it was not popped from IQ1. Note that `vertex_id` was explicitly loaded with *peek*, as opposed to `T2` and `T3`, where the task parameters are implicitly popped from their IQs by TSU (Section III-E). **T2** calculates the new distances to all the neighbors of `vertex_id` from the root using their edge values and sends this value to the owner of `T3` data. **T3** checks whether the distance of `neigh_id` from the root is smaller than the previously stored value. If so, `neigh_id` needs to be inserted in the frontier.

Dalorex does not use a barrier after each epoch to explore the new global frontier. Instead, each tile has a local frontier to allow for a continuous flow of tasks. The ***local frontier*** is a bitmap that accumulates the updates to the vertices that a tile owns. **T4** is responsible for re-exploring the local frontier. To avoid iterating over every 32-vertex block of the bitmap when `T4` is invoked, `T3` pushes the ID of a new block to be explored (`blk_id`) into `IQ4`. `T4` then reads from `IQ4` and pushes the vertices into `IQ1` so that they are processed again.

Remote invocation: When placing the parameters of the next task into a CQ, the first flit is the index of the distributed array to be accessed so that the message is routed to the tile containing the data for that index (details in Section III-E). Sending tasks to other tiles is akin to the non-blocking remote function calls employed by Tesseract [2]. Unlike Tesseract, Dalorex task invocations are non-interrupting when received.

Termination: The program ends when all tiles are idle. This is determined by aggregating a hierarchical, staged, idle signal from all the tiles (co-located with the clock and reset signals). Similar to a loosely-coupled accelerator [12], [51], the host gets an interrupt when the global idle signal is set to notify that the work is completed.

Synchronization: Although we strive to avoid synchronization between graph epochs, Dalorex also supports global synchronization by reusing the idle signal. To have synchronization per epoch, `T3` should not push new frontier vertices into the IQ of `T4` and simply add them to the bitmap frontier. When all vertices are processed, the host detects that the chip is idle, and it sends a message to all tiles to trigger `T4` and explore the new epoch. We characterize performance with and without epoch synchronization in Section V.

Host: The host is a commodity CPU that arranges the load of the program binary and the dataset from disk to the chip. The program, composed of tasks and memory-mapped software configurations, is distributed as a broadcast and is identical for all the allocated tiles. The data is distributed such that every tile receives an equal-sized chunk of each array.

Dalorex does not use virtual memory, although the host processor can still use virtual addressing within its memory. This avoids the overheads of address translation, which are exacerbated in graph applications due to irregular memory accesses [40]. Another advantage of having private, uncontested access to memory is not needing to deal with memory coherence or consistency issues.

D. Graph size vs. Chip size

Dalorex uses a homogeneous 2D layout of tiles. Because the size of a tile's local memory is determined at fabrication time, the aggregated chip storage scales linearly with the computing capacity (see Section V-B for the storage-to-compute ratio). Dalorex could be deployed on a $151mm^2$ chip, with 16×16 tiles and 512MB memory, or at wafer-scale, with 256×256 tiles and 128GB memory.

Processing Larger Graphs: Building on previous work on distributed graph processing [28], [63], we propose using graph partitioning to split larger graphs into multiple parts, where each part is computed on a separate Dalorex chip. The aggregated storage on-chip determines the maximum size of a graph partition. Smaller graphs can also be computed concurrently in rectangular subsets of tiles within a chip, uncontested, as they do not share hardware resources. The tiles that are not allocated to run programs are switched off.

Selecting the number of tiles a program runs on has a lower bound: it should run a subset of Dalorex that is large enough to fit the dataset into the aggregated memory capacity. It can also use a larger number of tiles to parallelize even further. Section V shows that Dalorex scales close to linearly until the parallelization limits are hit when a tile handles less than a thousand vertices.

E. Dalorex Hardware

Fig. 4 shows the structure of a tile in the Dalorex architecture. The Processing Unit (PU) is a very power- and area-efficient unit that resembles a single-issue in-order core but without a memory management unit or L1 cache, as the data is accessed directly from the scratchpad memory. The tile area is dominated by the scratchpad SRAM, which contains the data arrays, the code, and the input/output queue entries. The queues are implemented as circular FIFOs using the scratchpad. Queue sizes are configured at runtime based on the number of entries specified next to the task declaration (see Listing 1). A queue entry can be either 32 or 64 bits, depending on the chip's target total memory size. (A 32-bit Dalorex can process graphs of up to 2^{32} edges.)

The **Task Scheduling Unit** (TSU) is the key hardware unit of Dalorex's hardware-software co-design. It contains the task configurations and scheduling policy and handles the queues'

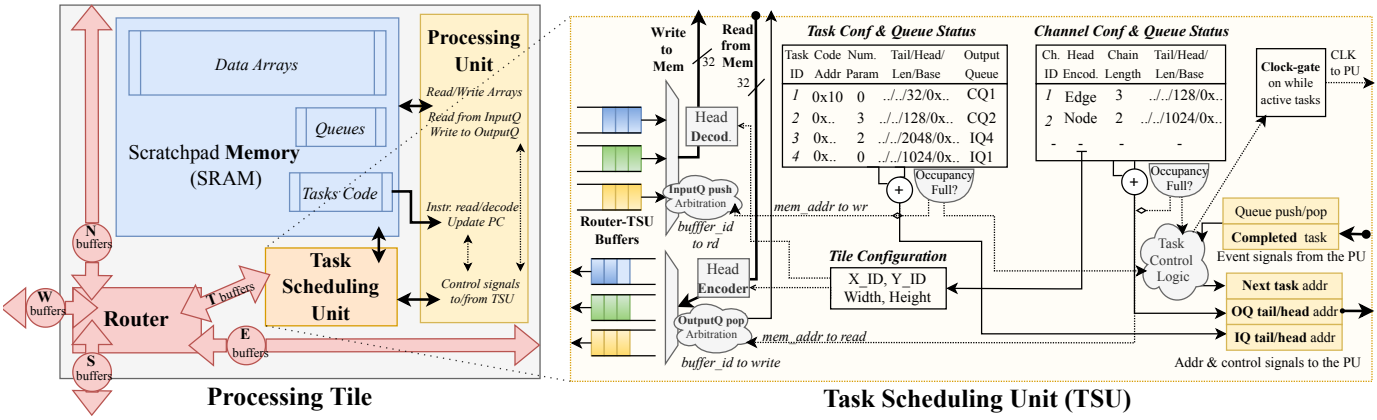


Fig. 4. Organization of a processing tile. TSU feeds the PU the next task to execute based on the occupancy of the task queues. The router connects TSU with the network. TSU uses SRAM to write incoming network data (push) to the IQs and reads (pop) outgoing data from the channel queues (CQs).

head and tail pointers. The TSU has a read-write port into the scratchpad for pushing data from the router buffers to the input queue (IQ).

Queue-specific registers: The tail and head pointers of queues are exposed to software through dedicated registers. This allows the PU to read or write from its queues with a register operation, avoiding address calculation. A read from this register results in a load from the scratchpad using the corresponding queue's head pointer provided by the TSU. This read also triggers the update of the head pointer in hardware at the *Task Queue Status* table.

Scheduling: TSU is responsible for invoking tasks based on the status of the queues, resulting in a closed-loop feedback system. Tasks cannot block; they execute from beginning to end. TSU may only invoke a task if its IQ is not empty and its OQ has sufficient free entries. When all IQs are empty, TSU disables the clock of the PU to save power. TSU needs to arbitrate when two or more tasks have non-empty IQs.

Priority: The queues' occupancy acts as a sensor for the TSU to decide which task to prioritize. A task has three priority modes based on queue occupancy: high priority if its IQ is nearly full, medium priority if its OQ is nearly empty, and low priority otherwise. When two or more tasks have high/medium priority, the one with a larger queue size takes precedence. Testing several static priorities and round-robin schemes, we found that the occupancy-based priority worked best because: (1) A large source of network contention is end-point back-pressure, so preventing full IQs eases contention; (2) Reaching high resource utilization relies on tiles giving each other work, so keeping OQs non-empty is beneficial. Using an occupancy-based priority order results in a feedback loop for task scheduling achieving high core utilization and low network contention. These heuristics are micro-coded as event-condition-action rules based on the reaction time to prioritize a task and prevent the IQ from getting full.

Channel queue: During program execution, when a task outputs the parameters for the next task to execute on another tile, a channel queue places the data into the network, which is delivered to its destination.

Network channel: It connects a channel queue with a task's IQ. Messages can be composed of several flits, each being a parameter of the task to be called. Network communication is composed of flits traveling in different logical channels that share the same network-on-chip (NoC). In our experiments, a flit has the same size as a queue entry, which is the width of the PU's ALU and the memory addresses (32 bits).

Router: It has bi-directional ports to north, south, east, west, and towards the tile's TSU. Routing is determined by the router based on the data in the first flit of a message, which we call the *head*. Since the head flit always contains a dataset-array index, and these arrays are statically distributed across the chip, this index is used to obtain the destination tile. The TSU's channel table contains the sizes of the local chunks and the number of parameters (flits) of each message type. The head encoder uses that information to calculate the destination tile and the local index (modulo the chunk size). The encoder also uses the width of the Dalorex grid to obtain the X/Y coordinates. Depending on the width and height of the chip, the upper bits of the head flit encode the destination tile ($log_2(width) + log_2(height)$). Routers compare incoming head flits with their local X/Y tile ID to determine where to route it next. If routed to the TSU, the head decoder removes the head flit's tile-index bits before pushing it to the IQ.

The payload-based routing saves network traffic as it does not use metadata. The length of messages at each channel is known, and its flits are always routed back to back since a route (from input to output port) opens with the first flit and closes after the corresponding number of flits has left the router. Interleaving flits from two messages going to the same output port on the same channel is not allowed. The router can route to different output ports simultaneously. However, it arbitrates by doing a round-robin between the messages from inputs ports that want to route to the same output port.

Channels buffers: In addition to identifying the task type, communicating tasks in different channels prevents deadlocks. Although channels might contend to use the NoC, the routers contain buffers per channel so that a clogged channel does not block others. Each router has a pool of buffer slots per outbound direction shared between the channels. The size of this pool is a tapeout parameter, but the number of buffer slots per channel is software-configurable, as are the sizes of the input/output queues.

F. Communication Network

Since Dalorex's programming model uses task invocations that do not return a value, communication is one way only, resembling a software pipeline. Therefore, the communication latency between the sender and receiver tiles does not contribute to the execution time if the pipeline is full, i.e., task invocations are continuous. However, the throughput would suffer if bubbles were formed due to network contention.

Graphs vertices have variable degrees. Some vertices—often referred to as hot vertices—have much higher vertex-degree than others. The channel buffers and queues mitigate the communication peaks caused by hot vertices and help keep the pipeline effect. *Dalorex uses low-order bits of indices to distribute data randomly, so the number of hot vertices per tile is relatively uniform.* Should the graph be sorted by vertex degree, we build the global CSR so that consecutive vertices fall into different tiles. This uniformity avoids excessive endpoint contention at the TSU with messages from the router.

We observed that the network topology could be another source of contention. Dalorex uses a 2D torus to avoid the contention towards the center we observed on a 2D mesh. Our 2D torus is a wormhole network with dimension-ordered routing. It also implements a local bubble routing to avoid the ring deadlock. This network can be fabricated with nearly equidistant wires by having consecutive logical tiles at a distance of two in the silicon. A 32-bit 2D torus is 50% bigger than a 2D mesh, but the torus provides twice the bisection bandwidth (BB) and 33% fewer hops [48].

Since we target the design of Dalorex to be scalable even with hundreds of tiles per dimension (tens of thousands of tiles), scalability of network utilization is part of the design goal. Scaling Dalorex up to the next power-of-two tiles per dimension results in four times the total tile count, but BB only doubles. While latency is greatly hidden by the pipeline effect, BB is critical for scalability since the vertex updates are irregular and result in communication to any tile. This disparity increases the contention with network sizes. To overcome that, we also explore ruche networks [27], [48].

Ruche networks are long physical wires that bypass routers. They increase the router radix and decrease the latency from source to destination. For example, in a network with a ruche factor of 2, a tile could route to its immediate neighbors or tiles at a distance of 2. A full ruche network of factor R increases the BB by a factor of $(R-1)\times$ over the underlying network, so increasing sizes of R with larger network sizes can compensate for the previously observed BB decrease. Section V provides a performance characterization of torus vs. mesh, with and without ruche networks.

G. Scalable Memory Bandwidth

Large scratchpads are more area-efficient with modern, smaller, FinFET transistor nodes [10], [16], [65]. This allows packing more SRAM on-chip than ever before. A real-world example is the Wafer-Scale integration of Cerebras, which enables 40GB of on-chip SRAM storage [36].

A dedicated scratchpad memory per tile enables immediate, uncontested access to memory. The width of each memory port is equal to the network width. The PU can make one memory read and one write per cycle. This is leveraged by instructions writing to queues from data arrays.

The PU has another port to fetch instructions from the scratchpad. While the scratchpad is heavily banked to save per-access energy, not all banks need to have all ports, e.g., the instruction port can exist only for a fraction of the local memory, setting a limit to the code size.

Since Dalorex's memory is distributed on-chip, the aggregated memory bandwidth increases linearly with the number of tiles, unlike many modern hardware architectures.

IV. Evaluation Methodology

In addition to four graph algorithms, we evaluated one sparse linear algebra kernel to demonstrate the generality of our approach for memory-bound applications.

We adapted the following **applications** from the GAP benchmark [6] and GraphIt [72], splitting the program into tasks at each indirect memory access: *Breadth-First Search (BFS)* determines the number of hops from a root vertex to all vertices reachable from it; *Single-Source Shortest Path (SSSP)* finds the shortest path from the root to each reachable vertex; *PageRank* ranks websites based on the potential flow of users to each page [34]; *Weakly Connected Components (WCC)* finds and labels each set of vertices reachable from one to all others in at least one direction (implemented using graph coloring [57]); *Sparse Matrix-Vector Multiplication (SPMV)* multiplies a sparse matrix with a dense vector.

Datasets: We used real-world networks and synthetic datasets. We use several different sizes of synthetic RMAT graphs [35] of up to 67M vertices (V) and 1.3B edges (E), with up to 12GB of memory footprint, as well as real-world graphs: LiveJournal (V=5.3M, E=79M), Wikipedia (V=4.2M, E=101M) and Amazon (V=262K, E=1.2M).

A. Simulation Approach

We built a C++ cycle-level simulator to evaluate Dalorex due to the novelty of the execution model and the simplicity of the architecture. Task instructions are modeled at each clock cycle; the network flits flow from tile to tile, from source to destination, one cycle per hop. Our simulations are validated to provide correct program outputs over sequential x86 executions of the applications we evaluate [6].

Current architectures pay an enormous energy cost associated with accessing off-chip memory [46]. Reading data locally consumes nearly two orders of magnitude less energy than moving an equivalent block of data across 15mm of on-chip wire [22] and three orders of magnitude less energy than bringing the data from off-chip DDR3 [62].

Power and Area Model: With the latest technology, SRAM can achieve very high densities in FinFET, ranging from 29.2 Mb/mm^2 in 7nm [65] to 47.6 Mb/mm^2 in 5nm [10]. New technologies like FinFET or CMOS-ULVR show that leakage power can get as low as 1.7nW per cell [32], 5μW per 8KB

(8192 bytes) macro [66], or 16.9μW for a 32KB macro at 7nm [65]. Although 5nm is achievable today [16], we will be using 7nm for our area and power models since this is a more mature transistor process, and we were able to find more reliable numbers for SRAM and logic. Reading a bank of data consumes 5.8pJ, and writing it 9.1pJ, with an access time of 0.82 ns [65]. Thus, we use a 1GHz frequency for Dalorex to fit the memory access time within the cycle length.

In addition to area-efficient SRAM technology, Dalorex uses processing units that resemble slim cores. We estimate the PU area considering the RISC-V Celerity, Snitch, and Ariane cores [15], [68], [70]. To determine the dynamic and leakage power of the single-issue in-order core, we use the reports from Ariane [67] and transistor power-scaling ratios to calculate the energy of those operations on a 7nm process [58], [64].

Regarding the NoC, we explore a 2D-Torus and a 2D-mesh, with and without ruche networks. These are modeled hop by hop between the routers, making the simulator precise for cycle count and energy. We use 8pJ as the energy to move a 32-bit flit one millimeter [41] and assume the energy of moving a flit at the router to be similar to an ALU operation. The NoC area is calculated based on Ou et al. [48]. These NoC options are explored in Fig. 8, and for other results Dalorex uses a regular torus NoC for grids up to 32×32 (1024 tiles) and a torus NoC with ruche channels for larger grids.

B. Comparison with the State-of-the-Art

We looked at the literature on graph accelerators, i.e., those with a specialized hardware pipeline to process the different stages of graph traversing, and focused on Polygraph [14]—the latest work. We evaluated the code that the authors kindly provided and experimentally confirmed that Polygraph's performance plateaus with configurations larger than 16 cores while Dalorex continues to scale. This is expected since that configuration already saturates the 512GB/s of DRAM bandwidth provided by their 8 memory controllers using High-Bandwidth Memory (HBM).

However, PIM-based proposals have a memory system that scales with the number of computing resources. From these works, we compare ourselves with Tesseract [2]. We have simulated a more recent PIM-based approach, GraphQ [76], which reported 3.3× performance and 1.8× energy improvements on average over Tesseract. However, despite communicating with the authors and using their methodology, we were unable to obtain the cycle and energy values from simulation outputs that matched their results. We do not include GraphQ in our discussion because we obtained 3.2× and 4.6× larger runtime and energy consumption (in geomean) than in [76].

We evaluate Tesseract using a Hybrid Memory Cube (HMC) configuration of 16 cores per cube (one per vault), aggregating a total of 256 cores among the 16 cubes. To match their core count, we use a 16×16 Dalorex grid with 4.2MB of memory per tile. To simulate Tesseract, we follow their methodology by using the Zsim simulator [55] with a 3D-memory power model [52].For the energy spent by the cores, we use the same power model as Dalorex—based on a 7nm transistor node. We compare runtime and energy performance for the duration of the graph processing time, not considering loading the dataset from disk to HMC or to the Dalorex chip. Although Tesseract already showed significant speedups over server-class OoO systems, we collected the cycle count of our x86 server while validating the output of graph applications to confirm that both Tesseract and Dalorex perform much better thanks to their scalable memory bandwidth (MBW).

V. Results

A. Improvements over the HMC-based prior work

This section demonstrates the gains in performance and energy efficiency of Dalorex over Tesseract [2], both using an equal number of cores (256). We break down the **impact of the different optimizations of Dalorex** by starting with Tesseract and then evaluating Dalorex by adding one feature at a time to gradually reach full Dalorex.

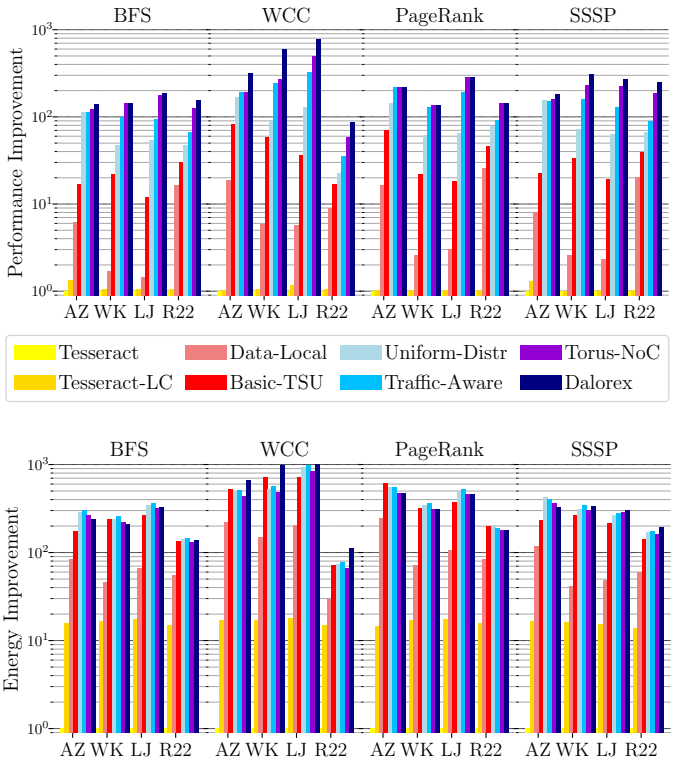


Fig. 5. Improvements in performance and energy efficiency achieved with the addition of large caches (LC) to Tesseract, and six Dalorex optimizations normalized to Tesseract. All configurations use 256 processing cores. X-axis studies four datasets. Y-axis is logarithmic, higher is better. Note that since PageRank necessitates per-epoch synchronization, the last datapoint (Dalorex-full) still uses a global barrier; thus, the bars do not change.

We provision Tesseract with a 2MB private cache per core (512 MB of aggregated cache) and remove DRAM background energy, to approximate the impact of using distributed SRAM on performance and energy ($Tesseract - LC$). We evaluate Dalorex without TSU by using the program flow of Tesseract (with remote vertex updates interrupting) to study the impact of vertex-based (Tesseract) versus Dalorex's array chunking and task splitting ($Data - Local$). We add a basic TSU to

invoke tasks with round-robin scheduling to evaluate the impact of non-blocking and non-interrupting communication (*Basic-TSU*). We distribute the data arrays by low-order bits instead of high-order bits to evaluate the impact of data placement. ($Uniform-distr$). We add the scheduling policy that prioritizes tasks based on queue occupancy ($Traffic-aware$). We use a 2D Torus instead of a 2D Mesh ($Torus-NoC$). Finally, we removed the global barrier synchronization after each epoch, reaching $Dalorex-full$. Section V-C further analyzes the impact of the NoC by testing one more design that benefits larger Dalorex grids.

Performance: Fig. 5 (top) shows that Dalorex substantially outperforms Tesseract across all datasets and applications. The dataset-chunking strategy and the data-local execution on Dalorex's homogeneous architecture improves performance by 6.2× ($Data-Local$). On top of that, non-blocking and non-interrupting task invocation through TSU improve performance by 4.7×, uniform data placement improves another 2.6×, and traffic-aware scheduling 1.7× more. Finally, removing the barriers and upgrading the NoC provides an extra 1.8×, compounding a 221× geomean improvement. Synchronization in graph workloads causes each epoch to take as long as the slowest tile's execution, increasing the total runtime. WCC, having more epochs, benefits the most from barrierless processing. Similarly, Wikipedia (WK) benefits most from removing the barrier as its graph structure leads to more epochs.

Energy: Fig. 5 (bottom) shows that the performance improvement of Dalorex over Tesseract, together with its power-efficient design, yields a very large improvement in energy consumption. The compound improvements from using SRAM technology (16×), our architecture and data layout (5.2×), and TSU (3.9×), total a geomean of 325×. The breakdown shows that the energy of refreshing DRAM has the biggest impact on Tesseract (also concluded in their paper). Upgrading the NoC to a 2D Torus increases energy usage by 12% geomean, besides being 40% faster. This is because the Torus network is more power-hungry and has longer wires.

Area: The 16×16 Dalorex with a 4.2MB memory per tile uses much less chip area ($305mm^2$) than the aggregated area of the 16 cubes of Tesseract ($3616mm^2$). The prior work based on HMC is constrained by the number of cores that can be integrated into the logic die of a memory cube—often one per vault. Since a cube contains 8GB, a core accesses a 512MB DRAM vault. Most of this DRAM space is unused with the datasets that Tesseract evaluated (the empty bitlines of DRAM are switched off to save power). However, larger datasets would have a much bigger runtime in the HMC-based design since it cannot have more cores without increasing the number of memory cubes.

HMC is also limited in scalability due to the large *power density*, which makes it more challenging to cool in 3D integrations [75]. In Dalorex, power is evenly distributed, so power density stays below $300mW/mm^2$ for all our experiments. This is much below the limits of air-cooled 2D chips ($\sim 1.5W/mm^2$ [73]). Another limitation of the HMC-based approaches is that lower inter-cube *communication bandwidth* made their authors consider graph partitioning per cube, limiting the scalability of each partition.

To summarize, such large across-the-board improvements are achieved with a conjunction of optimizations at different levels of the software-hardware stack. Dalorex: (1) reduces data movement with its *fully* data-local program execution model, where only task parameters are moved; (2) task invocations are natively supported through the TSU, so there are no interrupt overheads as in prior remote procedure calls; (3) task are scheduled based on the network's task traffic; (4) does not require barrier synchronization between tiles for BFS, WCC, and SSSP; (5) decouples the placement of vertices and edges, giving an equal number of edges to each tile, improving work balance; (6) utilizes SRAM to store the tile-distributed dataset, making data access immediate and energy-efficient for the challenging fine-grain irregular accesses. Also, SRAM is offered in finer sizes than DRAM. This enables Dalorex to use a few megabytes of memory per tile, which we found to be energy-optimal in the scaling experiments of the next section.

B. Dalorex Scales Beyond Thousands of Tiles

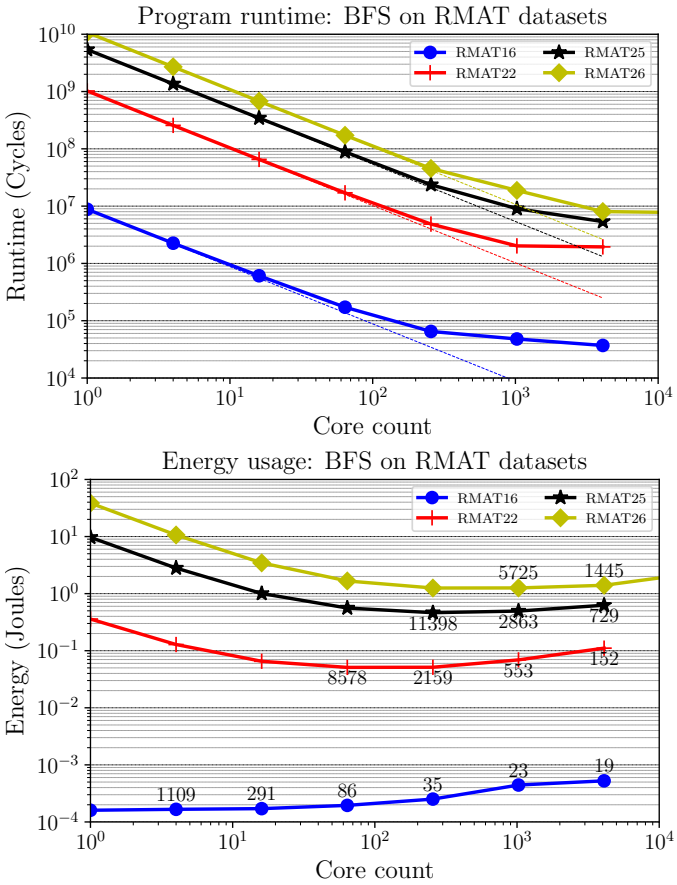


Fig. 6. Analysis of the runtime (cycles) and energy consumption (Joules) of BFS for four RMAT datasets and scaling core counts.

Fig. 6 evaluates strong scaling for BFS with increasing sizes of Dalorex grids, and four RMAT datasets of size 2^{16}, 2^{22}, 2^{25} and 2^{26} vertices (average ten edges per vertex).

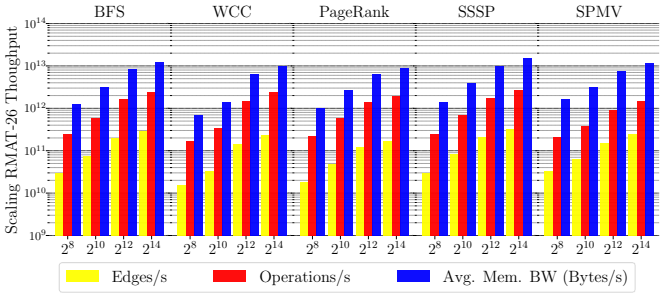


Fig. 7. Throughput in terms of executed instructions and edges per second, and the average on-chip memory bandwidth (MBW) used to achieve that. The X-axis is the size of the Dalorex grid used when analyzing strong scaling RMAT-26, ranging from 256 to 16,384 tiles. The Y-axis is logarithmic.

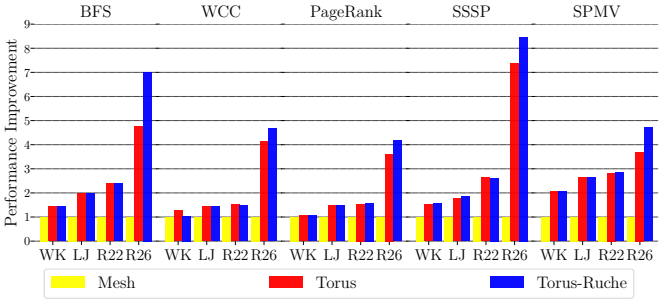


Fig. 8. Performance improvement of Torus and Torus-Tuche over Mesh. The X-axis shows the datasets used for each of the applications evaluated. RMAT-26 runs on 64×64 tiles, and the rest 16×16.

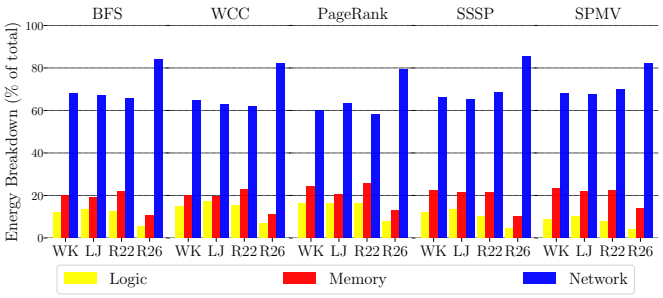


Fig. 9. Breakdown of the energy consumed by the computing logic, the SRAM cells, and the network communication (including routing and wire energy). The Y-axis is the percentage of total energy spent running the program. The X-axis shows the datasets used for each of the applications evaluated. RMAT-26 runs on 64×64 tiles, and the rest 16×16.

Performance Scaling: The upper plot of Fig. 6 shows the runtime (in cycles) of executing each dataset on Dalorex with an increasing number of tiles by multiples of four. The most exciting result of this analysis is that Dalorex has a close to linear scaling until it hits the parallelization limit. This occurs when the chunk of data per tile is ∼1,000 vertices, indicating that performance is not limited by MBW (as is the case for all previous work) but tiles starving for work.

Energy Scaling: To understand the optimal scratchpad size for energy consumption, we analyzed energy as the number of tiles increases and the required scratchpad size decreases. Each scratchpad stores the dataset chunk, the program binary, and the queues. Fig. 6 (bottom) shows the total energy spent to run BFS for different numbers of tiles. Next to each datapoint is the memory used by each tile (in KB).

As the tile count increases and the dataset chunk per tile becomes smaller, energy first decreases to a minimum and then increases. This is mainly driven by leakage power. Since the aggregated memory capacity remains nearly constant in this experiment, the SRAM leakage power becomes less significant with larger parallelizations. Total energy keeps decreasing while the cores are fully utilized. The deflection point, i.e., minimal energy execution for each dataset, arrives when the amount of data per tile is ∼10,000 vertices. This optimal range is invariant with dataset size. The smallest dataset (RMAT-16) reaches this point very early, being already past that point with 64 tiles, where each tile holds ∼1,000 vertices.

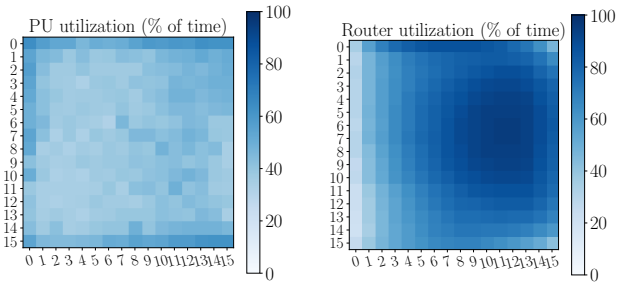
Fig. 6 shows that the performance keeps scaling beyond the energy-optimal configuration. Unlike clusters, where performance degrades with small messages between computing nodes, Dalorex communication is at a word granularity. Scaling is not limited by MBW but only when the workload per tile is so low that we lose the pipeline effect and the PUs starve for tasks. While approaching the parallelization limit of a dataset, energy increases because of the leakage energy of the additional PUs that are not fully utilized.

Throughput: Fig. 7 shows the number of edges and operations processed per second as a measure of the throughput and the average aggregated MBW by all tiles. We study this with doubling counts on X- and Y-dimensions for all benchmarks running our biggest RMAT dataset. We observe that both throughput and utilized MBW grow until the largest configuration we simulated: 128×128 (2^{14}, i.e., 16,384 tiles). This last configuration reaches two teraoperations/s, using over ten terabytes/s of MBW. The throughput and MBW of Dalorex for graph applications are beyond the reach of prior work in hardware accelerators using HBM and the 3D-memory integration that Tesseract proposed. Dalorex MBW is possible by having all the memory storage distributed across tiles and linearly increasing the number of memory ports with the tile count. The fact that the MBW does not saturate allows the PUs to process vertices and edges at the very high rates shown in Fig. 7. Note that we reported the average MBW of the whole program and not the maximum utilized at a given time. The peak MBW available only increases with the number of tiles, which is 131TB/s on a 128×128 Dalorex.

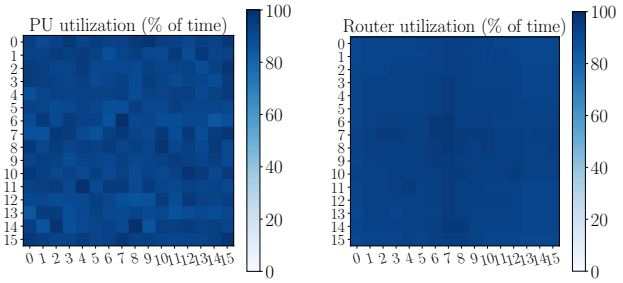
C. Characterizing the Network-on-Chip (NoC)

We analyze Dalorex with the three NoC types we considered in Section III-F: 2D mesh, 2D torus, and combining the torus with ruche channels. Fig. 8 shows the performance improvement of both torus options over the mesh. Using a 16×16 torus is nearly twice as fast as a mesh, on average, for the smaller datasets (Wikipedia, LiveJournal, and RMAT-22), which justifies the area cost of an additional 0.2% of the total chip area (using 4MB tiles).

For this 16×16 grid, we generated heatmaps of the utilization of PUs and routers to visually demonstrate the advantage of torus over mesh. Fig. 10 shows that the contention towards the center of the mesh (top) clogs the NoC and makes the



(a) 2D Mesh



(b) 2D Torus network

Fig. 10. Heatmaps of the utilization of PUs and routers as a percentage of runtime while running SSSP on RMAT-22. The 16×16 tiles are connected by a mesh (top) or torus (bottom).

PUs starve for tasks, whereas the torus (bottom) has a uniform router utilization, unleashing the full potential of the PUs.

As shown in Fig. 8, the improvement of the torus NoC becomes even greater for the 64×64 grid used to evaluate RMAT-26. We found *ruche with torus* to only improve performance on the large grid. Despite that the ruche-torus NoC uses more than twice the area of a regular torus (extra cost of 1.2% area) and higher power, the performance gains at the 64×64 grid justify its use. Although not shown here, we have found ruche combined with mesh not to be as effective as torus alone, despite its larger area cost.

Fig. 9 breaks down the energy consumed by Dalorex. We use a 16×16 grid to run WK, LJ, and RMAT-22, and 64×64 to run RMAT-26. In Dalorex, the network consumes the most energy: the larger the network, the longer the average distance traveled to update a vertex, therefore, a greater share of the total energy consumption. This is expected because Dalorex uses energy-efficient memories and very simple processing units (PU). PUs are also not actively waiting for messages to arrive but are powered off by the TSU when idle.

VI. Further related work & Discussion

Due to storage limitations, graph networks with trillions of edges inevitably need to be partitioned and processed by multiple systems. However, within each system (where each partition is processed), distributed graph processing frameworks [3], [38], [63], [74] are bottlenecked by the memory hierarchies in current computer architectures.

Dalorex, and other HMC-based approaches for graph processing [2], [71], [76] behave as a large accelerator for the host CPU. Loosely-coupled accelerators for graph applications have been investigated before, e.g., [14], [19], [49]. However, ASIC accelerators cannot execute the variety of workloads that ISA-programmable processors can. Dalorex is not restricted to executing graph applications and can be used for other domains since it is software programmable.

Unlike the prior work, which assumes an HMC technology that has not been adopted, there is already an example of an architecture where all the memory is distributed on a chip that is being commercialized by Cerebras [9], [36] to speed up Machine Learning applications.

As we show in this paper, the Dalorex execution model and design decisions like network type, local-memory size, and task scheduling unleash an outstanding graph processing performance for distributed-memory architectures.

Another example of an architecture that could use Dalorex is a regular memory hierarchy with large L1 caches per tile, leveraging prior work in cache-scratchpad duality [11], [13], [31] to provide a hybrid solution that would benefit both cache-averse and cache-friendly workloads.

VII. Conclusion

Dalorex is designed to massively parallelize applications that are memory-bound due to the irregular memory accesses, for which prior work does not exhibit good strong scaling. Datasets larger than the aggregated memory capacity of a Dalorex chip could be partitioned across several chips, having each subproblem parallelized to the extreme.

Since Dalorex is ISA-programmable, it is applicable to any application fitting in the task-based programming model, although it is most advantageous for those bottlenecked by pointer indirection or atomic operations, e.g., SPMV.

This paper demonstrates strong scaling on four graph algorithms and one sparse linear algebra kernel. We found that reaching the parallelization limits requires: (1) a *uniform work balance*, which we achieve with an equal amount of data per tile; (2) architectural and programming *support to invoke remote tasks natively* with no message overhead; (3) a *traffic-aware* arbitration of tasks; (4) a *NoC that minimizes contention*, where a torus is superior to the mesh; and (5) a *NoC that scales bisection bandwidth* with very large tile counts on a 2D silicon by adding ruche channels to the torus. Together with the data partitioning scheme that allows barrierless frontiers, these optimizations make Dalorex two orders of magnitude faster and more energy-efficient than the state-of-the-art in PIM-based graph processing.

Acknowledgments

This material is based on research sponsored by the Air Force Research Laboratory (AFRL), Defense Advanced Research Projects Agency (DARPA) under agreement FA8650-18-2-7862, and National Science Foundation (NSF) award No. 1763838. [1] We thank Tyler Sorensen for his useful feedback.

[1]The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF, AFRL and DARPA or the U.S. Government.

REFERENCES

[1] M. Abeydeera and D. Sanchez, "Chronos: Efficient speculative parallelism for accelerators," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1247–1262.

[2] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 105–117.

[3] "Apache Giraph," The Apache Software Foundation, http://giraph.apache.org/.

[4] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff, "OpenPiton: An open source manycore research framework," in *ASPLOS*. ACM, 2016, pp. 217–232.

[5] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–10.

[6] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[7] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, "To push or to pull: On reducing communication and synchronization in graph computations," *CoRR*, vol. abs/2010.16012, 2020. [Online]. Available: https://arxiv.org/abs/2010.16012

[8] E. G. Boman, K. D. Devine, and S. Rajamanickam, "Scalable matrix computations on large scale-free graphs using 2d graph partitioning," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[9] Cerebras Systems Inc., "The second generation wafer scale engine," https://cerebras.net/wp-content/uploads/2021/04/Cerebras-CS-2-Whitepaper.pdf.

[10] T.-Y. J. Chang, Y.-H. Chen, W.-M. Chan, H. Cheng, P.-S. Wang, Y. Lin, H. Fujiwara, R. Lee, H.-J. Liao, P.-W. Wang, G. Yeap, and Q. Li, "A 5-nm 135-Mb SRAM in EUV and high-mobility channel finfet technology with metal coupling and charge-sharing write-assist circuitry schemes for high-density and low-vmin applications," *IEEE Journal of Solid-State Circuits*, vol. 56, no. 1, pp. 179–187, 2021.

[11] C. C. Chou, A. Jaleel, and M. K. Qureshi, "Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 1–12.

[12] E. Cota, G. D. Guglielmo, P. Mantovani, and L. Carloni, "An analysis of accelerator coupling in heterogeneous architectures," in *Proceedings of the 52nd Design Automation Conference (DAC)*, 2015.

[13] E. G. Cota, P. Mantovani, and L. P. Carloni, "Exploiting private local memories to reduce the opportunity cost of accelerator integration," in *Proceedings of the 2016 International Conference on Supercomputing*, 2016, pp. 1–12.

[14] V. Dadu, S. Liu, and T. Nowatzki, "Polygraph: Exposing the value of flexibility for graph processing accelerators," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 595–608.

[15] S. Davidson, S. Xie, C. Torng, K. Al-Hawai, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, A. Amarnath, B. Veluri, P. Gao, A. Rao, G. Liu, R. K. Gupta, Z. Zhang, R. Dreslinski, C. Batten, and M. B. Taylor, "The celerity open-source 511-core risc-v tiered accelerator fabric: Fast architectures and design methodologies for fast chips," *IEEE Micro*, vol. 38, no. 2, pp. 30–41, 2018.

[16] S. Enjapuri, D. Gujjar, S. Sinha, R. Halli, and M. Trivedi, "A 5nm wide voltage range ultra high density sram design for l2/l3 cache applications," in *2021 34th International Conference on VLSI Design and 2021 20th International Conference on Embedded Systems (VLSID)*, 2021, pp. 151–156.

[17] Esperanto Technologies, "Esperanto's et-minion on-chip risc-v cores," https://www.esperanto.ai/technology/.

[18] T. J. Ham, J. L. Aragón, and M. Martonosi, "DeSC: Decoupled supply-compute communication management for heterogeneous architectures," in *MICRO*. ACM, 2015.

[19] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proceedings of the 49th Annual International Symposium on Microarchitecture*, ser. MICRO, 2016. [Online]. Available: https://doi.org/10.1109/MICRO.2016.7783759

[20] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 9, pp. 950–961, 2015.

[21] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[22] R. Ho, K. W. Mai, and M. A. Horowitz, "The future of wires," *Proceedings of the IEEE*, vol. 89, no. 4, pp. 490–504, 2001.

[23] H. Hoffmann, "Stream algorithms and architecture," Ph.D. dissertation, Massachusetts Institute of Technology, 2003.

[24] "Hybrid Memory Cube (HMC)," http://www.hybridmemorycube.org, 2018.

[25] K. T. Johnson, A. R. Hurson, and B. Shirazi, "General-purpose systolic arrays," *Computer*, vol. 26, no. 11, pp. 20–31, 1993.

[26] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[27] D. C. Jung, S. Davidson, C. Zhao, D. Richmond, and M. B. Taylor, "Ruche networks: Wire-maximal, no-fuss nocs: Special session paper," in *2020 14th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*. IEEE, 2020, pp. 1–8.

[28] G. Karypis and V. Kumar, "Metis: A software package for partitioning unstructured graphs," *Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Version*, vol. 4, no. 0, 1998.

[29] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–29, 2017.

[30] S. Knowles, "Graphcore," in *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE, 2021, pp. 1–25.

[31] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon, "Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2012, pp. 376–388.

[32] T. S. Kumar and S. L. Tripathi, "Process evaluation in finfet based 7t sram cell," *Analog Integrated Circuits and Signal Processing*, pp. 1–7, 2021.

[33] H. T. Kung and C. E. Leiserson, "Systolic arrays for (vlsi)." Carnegie-Mellon University, Tech. Rep., 1978.

[34] P. Lawrence, B. Sergey, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," Stanford University, Technical Report, 1998.

[35] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *Journal of Machine Learning Reseach (JMLR)*, vol. 11, pp. 985–1042, Mar. 2010.

[36] S. Lie, "Multi-million core, multi-wafer ai cluster," in *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE Computer Society, 2021, pp. 1–41.

[37] E. Lockerman, A. Feldmann, M. Bakhshalipour, A. Stanescu, S. Gupta, D. Sanchez, and N. Beckmann, "Livia: Data-centric computing throughout the memory hierarchy," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 417–433. [Online]. Available: https://doi.org/10.1145/3373376.3378497

[38] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135–146.

[39] A. Manocha, T. Sorensen, E. Tureci, O. Matthews, J. L. Aragón, and M. Martonosi, "Graphattack: Optimizing data supply for graph applications on in-order multicore architectures," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 4, pp. 1–26, 2021.

[40] A. Manocha, Z. Yan, E. Tureci, J. L. Aragón, D. Nellans, and M. Martonosi, "The implications of page size management on graph analytics," in *International Symposium on Workload Characterization (IISWC)*, 2022, pp. 1–14.

[41] M. McKeown, A. Lavrov, M. Shahrad, P. J. Jackson, Y. Fu, J. Balkind, T. M. Nguyen, K. Lim, Y. Zhou, and D. Wentzlaff, "Power and energy characterization of an open source 25-core manycore processor." in *HPCA*, 2018, pp. 762–775.

[42] G. E. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[43] A. Mukkara, N. Beckmann, and D. Sanchez, "PHI: Architectural Support for Synchronization-and Bandwidth-Efficient Commutative Scatter Updates," in *Proceedings of the 52nd annual IEEE/ACM international symposium on Microarchitecture (MICRO-52)*, October 2019.

[44] Q. M. Nguyen and D. Sanchez, "Pipette: Improving core utilization on irregular applications through intra-core pipeline parallelism," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 596–608.

[45] Q. M. Nguyen and D. Sanchez, "Fifer: Practical acceleration of irregular applications on reconfigurable architectures," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1064–1077. [Online]. Available: https://doi.org/10.1145/3466752.3480048

[46] T. M. Nguyen and D. Wentzlaff, "Morc: A manycore-oriented compressed cache," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 76–88.

[47] M. Orenes-Vera, A. Manocha, J. Balkind, F. Gao, J. L. Aragón, D. Wentzlaff, and M. Martonosi, "Tiny but mighty: designing and realizing scalable latency tolerance for manycore socs." in *ISCA*, 2022, pp. 817–830.

[48] Y. Ou, S. Agwa, and C. Batten, "Implementing low-diameter on-chip networks for manycore processors using a tiled physical design methodology," in *2020 14th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*. IEEE, 2020, pp. 1–8.

[49] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 166–177, 2016.

[50] J. T. Pawlowski, "Hybrid memory cube (hmc)," in *2011 IEEE Hot Chips 23 Symposium (HCS)*. IEEE, 2011, pp. 1–24.

[51] L. Piccolboni, P. Mantovani, G. Di Guglielmo, and L. P. Carloni, "Broadening the exploration of the accelerator design space in embedded scalable platforms," in *HPEC*. IEEE Press, 2017.

[52] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "Ndc: Analyzing the impact of 3d-stacked memory+ logic devices on mapreduce workloads," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2014, pp. 190–200.

[53] S. Rahman, N. Abu-Ghazaleh, and R. Gupta, "Graphpulse: An event-driven hardware accelerator for asynchronous graph processing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 908–921.

[54] K. Rupp, "42 Years of Microprocessor Trend Data," https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/, 2018.

[55] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," *ACM SIGARCH Computer architecture news*, vol. 41, no. 3, pp. 475–486, 2013.

[56] K. S. Shim, M. Lis, M. H. Cho, O. Khan, and S. Devadas, "System-level optimizations for memory access in the execution migration machine (em2)," *CAOS*, 2011.

[57] G. M. Slota, S. Rajamanickam, and K. Madduri, "BFS and coloring-based parallel algorithms for strongly connected components and related problems," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*. IEEE Computer Society, 2014, pp. 550–559. [Online]. Available: https://doi.org/10.1109/IPDPS.2014.64

[58] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of cmos device performance from 180 nm to 7 nm," *Integration*, vol. 58, pp. 74–81, 2017.

[59] N. Talati, K. May, A. Behroozi, Y. Yang, K. Kaszyk, C. Vasiladiotis, T. Verma, L. Li, B. Nguyen, J. Sun, J. M. Morton, A. Ahmadi, T. Austin, M. O'Boyle, S. Mahlke, T. Mudge, and R. Dreslinski, "Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 654–667.

[60] M. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal, "Scalar operand networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 2, pp. 145–162, 2005.

[61] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee *et al.*, "The raw microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE micro*, vol. 22, no. 2, pp. 25–35, 2002.

[62] M. Technology, "Ddr3/ddr4 system-power calculator," https://www.micron.com/support/tools-and-utilities/power-calc.

[63] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From" think like a vertex" to" think like a graph"," *Proceedings of the VLDB Endowment*, vol. 7, no. 3, pp. 193–204, 2013.

[64] Q. Xie, X. Lin, Y. Wang, S. Chen, M. J. Dousti, and M. Pedram, "Performance comparisons between 7-nm finfet and conventional bulk cmos standard cell libraries," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 8, pp. 761–765, 2015.

[65] Y. Yokoyama, M. Tanaka, K. Tanaka, M. Morimoto, M. Yabuuchi, Y. Ishii, and S. Tanaka, "A 29.2 mb/mm2 ultra high density sram macro using 7nm finfet technology with dual-edge driven wordline/bitline and write/read-assist circuit," in *2020 IEEE Symposium on VLSI Circuits*, 2020, pp. 1–2.

[66] H. Yoshida, Y. Shiotsu, D. Kitagata, S. Yamamoto, and S. Sugahara, "Ultralow-voltage retention sram with a power gating cell architecture using header and footer power-switches," *IEEE Open Journal of Circuits and Systems*, vol. 2, pp. 520–533, 2021.

[67] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, Nov 2019, https://github.com/openhwgroup/cva6.

[68] F. Zaruba and L. Benini, "Ariane: An open-source 64-bit RISC-V application class processor and latest improvements," 2018, technical talk at the RISC-V Workshop https://www.youtube.com/watch?v=8HpvRNh0ux4.

[69] F. Zaruba, F. Schuiki, and L. Benini, "Manticore: A 4096-core risc-v chiplet architecture for ultraefficient floating-point computing," *IEEE Micro*, vol. 41, no. 2, pp. 36–42, 2020.

[70] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, "Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads," *IEEE Transactions on Computers*, 2020.

[71] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "Graphp: Reducing communication for pim-based graph processing with efficient data partition," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 544–557.

[72] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: A high-performance graph dsl," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–30, 2018.

[73] P. Zhou, J. Hom, G. Upadhya, K. Goodson, and M. Munch, "Electro-kinetic microchannel cooling system for desktop computers," in *Twentieth Annual IEEE Semiconductor Thermal Measurement and Management Symposium (IEEE Cat. No. 04CH37545)*. IEEE, 2004, pp. 26–29.

[74] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 301–316.

[75] Y. Zhu, B. Wang, D. Li, and J. Zhao, "Integrated thermal analysis for processing in die-stacking memory," in *Proceedings of the Second International Symposium on Memory Systems*, 2016, pp. 402–414.

[76] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "Graphq: Scalable pim-based graph processing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 712–725.