Squire: A General-Purpose Accelerator to Exploit Fine-Grain Parallelism on Dependency-Bound Kernels

Rubén Langarita
Barcelona Supercomputing Center
Barcelona, Spain
ruben.langarita@bsc.es

Jesús Alastruey-Benedé
Universidad de Zaragoza
Zaragoza, Spain
jalastru@unizar.es

Pablo Ibáñez-Marín
Universidad de Zaragoza
Zaragoza, Spain
imarin@unizar.es

Santiago Marco-Sola
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
Barcelona, Spain
santiago.marco@bsc.es

Miquel Moretó
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
Barcelona, Spain
miquel.moreto@bsc.es

Adrià Armejach
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
Barcelona, Spain
adria.armejach@bsc.es

***Abstract*—Multiple HPC applications are often bottlenecked by compute-intensive kernels implementing complex dependency patterns (data-dependency bound). Traditional general-purpose accelerators struggle to effectively exploit fine-grain parallelism due to limitations in implementing convoluted data-dependency patterns (like SIMD) and overheads due to synchronization and data transfers (like GPGPUs). In contrast, custom FPGA and ASIC designs offer improved performance and energy efficiency at a high cost in hardware design and programming complexity and often lack the flexibility to process different workloads.**

We propose Squire, a general-purpose accelerator designed to exploit fine-grain parallelism effectively on dependency-bound kernels. Each Squire accelerator has a set of general-purpose low-power in-order cores that can rapidly communicate among themselves and directly access data from the L2 cache. Our proposal integrates one Squire accelerator per core in a typical multicore system, allowing the acceleration of dependency-bound kernels within parallel tasks with minimal software changes.

As a case study, we evaluate Squire's effectiveness by accelerating five kernels that implement complex dependency patterns. We use three of these kernels to build an end-to-end read-mapping tool that will be used to evaluate Squire. Squire obtains speedups up to 7.64× in dynamic programming kernels. Overall, Squire provides an acceleration for an end-to-end application of 3.66×. In addition, Squire reduces energy consumption by up to 56% with a minimal area overhead of 10.5% compared to a Neoverse-N1 baseline.

***Index Terms*—Hardware accelerator, General-purpose, Fine-grain parallelism, Dynamic programming, Genomics**

I. Introduction

Modern multi-core architectures and accelerators have become the cornerstone for accelerating many workloads in scientific computing and engineering [1]. Many efforts have been made to accelerate HPC applications on modern hardware architectures such as CPUs and GPUs, as well as FPGA and custom accelerators (ASICs) for specific workloads [2]. Hence, HPC platforms are increasingly sought after to handle large-scale workloads that exploit different levels of parallelism available in the accelerators.

However, there is an emergent class of workloads that cannot fully exploit the massively parallel capabilities of mainstream accelerators. Many HPC applications are often bottlenecked by the execution of sequential workflows composed of rather small compute-intensive kernels that implement complex dependency patterns (dependency-bound). This is particularly noticeable in life science and healthcare applications, which implement long workflows of data-processing kernels [3]. Often based on stencil and Dynamic Programming (DP) computations, *dependency-bound* kernels tend to be moderate in size and implement complex data-dependency patterns that ultimately restrict parallelism exploitation.

Traditionally, coarse-grain parallelism approaches seek to execute independent kernels concurrently (inter-task) and improve resource utilization [4], [5]. However, offloading dependency-bound kernels to specialized hardware accelerators seldom pays off. Due to its rather small size, offloading workloads to decoupled accelerators often incur significant overheads both from data-transfer initialization and movement [6]. Similarly, task-level parallelization approaches often introduce synchronization delays that hinder performance. Also, load imbalance and data sparsity present significant challenges to the flexibility of conventional accelerators, ultimately limiting their performance [7].

Similarly, fine-grain (intra-task) parallelism is difficult to exploit in dependency-bound workloads. Complex dependency patterns and irregular computations make the exploitation of the underlying parallelism challenging [8], [9]. Moreover, the need for fine-grain synchronization introduces significant overheads during execution. Ultimately, these kernels often cannot saturate computing resources and effectively exploit fine-grain parallelism [10], [11].

Mainstream SIMD-based and GPU approaches are limited in accelerating these types of workloads. SIMD approaches face significant limitations in handling sparse data-processing tasks, such as gather and scatter operations, which introduce substantial latency overheads [12]. Additionally, SIMD instructions often struggle to implement complex data dependency patterns, limiting their effectiveness in many scenarios. Similarly, GPUs tend to provide only modest performance improvements on these small-scale workloads, as they cannot saturate GPU's compute resources (i.e., few threads and thread blocks). In addition, GPUs load imbalance and data-transfers overheads limit the performance benefits of offloading these types of kernels [6].

In contrast, custom FPGA-based and domain-specific ASIC designs offer improved performance and energy efficiency at the cost of complex and expensive hardware design, development, and fabrication processes [13]. In some cases, they also suffer from data transfer and synchronization overheads. Ultimately, custom hardware approaches often lack the flexibility to adapt to different workloads and incur significant programming and maintenance costs [14]–[16].

To address these challenges, our **goal** in this work is to enable (i) efficient exploitation of fine-grain parallelism in dependency-bound kernels, (ii) reducing data-transfers overheads, and (iii) providing hardware support for fast synchronization primitives.

In this work, we propose Squire, a general-purpose accelerator designed to effectively exploit fine-grain parallelism on dependency-bound kernels. Squire is equipped with several general-purpose in-order cores, called workers, and a hardware semaphore for rapid synchronization among the workers. Our proposal incorporates one Squire per core in a typical multicore system, connecting it to the memory hierarchy to directly access the virtual memory space. Each core controls one Squire, rapidly offloading workloads whenever it needs. Since workers share the same Instruction Set Architecture (ISA) as the core, we can develop and compile code for Squire just as we do for the core.

We discuss three potential use cases for our accelerator: data sorting, genomics, and signal processing. We examine some representative kernels and show how fine-grain parallelism can be exploited. These kernels include Radix Sort [17], Seeding [18], Chain [18], Smith-Waterman [19], [20], and Dynamic Time Warping [21].

This work makes the following contributions.

- We propose Squire, a general-purpose accelerator for exploiting fine-grain parallelism on dependency-bound kernels.
- We select five dependency-bound kernels and analyze them. Then, we adapt these kernels to be executed with Squire.
- We use three of these kernels to build an end-to-end read-mapping tool that will be used to evaluate Squire.
- We show how Squire can speed-up the five evaluated kernels. We also evaluate the end-to-end read-mapper to understand how Squire improves a full application. Finally, we perform an area and energy consumption study.

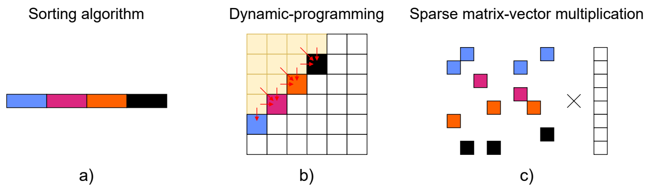

Fig. 1: Examples of coarse-grain tasks with fine-grain parallelism: (a) sorting, (b) dynamic programming matrix, (c) sparse matrix-vector multiplication. Each color in the data structures represents a chunk of fine-grain work.

Key results. Squire obtains speedups up to 7.64× in dynamic programming kernels. Overall, Squire provides an acceleration for an end-to-end application of 3.66×. In addition, Squire reduces power consumption by up to 56% with a minimal area overhead of 10.5% compared to a Neoverse-N1 baseline.

II. Motivation

Coarse-grain parallelism is desirable in high-performance computing environments to hide the overheads associated with task management and data movement. This inter-task parallelism means that each processing core operates on an independent task [4], [5], [18], [22]–[26]. However, this approach often struggles to exploit the fine-grain parallelism inherent in many kernels with complex dependencies.

Intra-task fine-grain parallelism is usually tackled via *SIMD* on general-purpose processors or *SIMT* on GPUs. However, these techniques are inefficient when targeting specific algorithms containing dependencies or sparse patterns. Well-known algorithms that suffer such problems include: Quicksort [27], Dynamic Time Warping [21], and Smith-Waterman [19], [20]. Additionally, data structures with sparse memory patterns, such as FM-Index [28]–[30], hash tables [17], and sparse matrix-vector multiplication (SpMV) [31], are often limited by the amount of memory-level parallelism that can be exposed. All these patterns are present in many applications.

Figure 1 highlights three kernels that exhibit fine-grain parallelism (shown using different colors) which is challenging to exploit due to existing dependencies. Figures 1a and 1c show how sorting and SpMV coarse-grain tasks could be further parallelized by processing chunks of the array or independent rows of the matrix in parallel. However, this is not efficient due data-dependent irregular patterns and the fact that SIMD gather/scatter memory operations are not efficient [12]. Subfigure 1b shows how parallelism is present per cell in a dynamic programming matrix. For some dynamic programming problems, antidiagonal vectorization is the best way to avoid dependencies; however, it requires data rearranging that diminishes potential gains. Support for exploiting this fine-grain parallelism can benefit a large set of workloads.

GPGPUs have also been proposed to tackle dynamic programming kernels [8], [32]. However, GPGPUs are designed for massive parallel workloads, and dependencies and sparsity hinder performance. In addition, offloading fine-grain parallel workloads is not recommended due to the high transfer time, and fine-grain synchronization is also challenging. For these reasons, GPGPUs obtain modest speed-ups when targeting dynamic programming algorithms [9]–[11], [33], [34]. Finally, custom hardware solves dependency constraints at the cost of fixing the functionality of the proposed components, losing generality [14]–[16], [35], [36].

Chain is a dynamic programming algorithm widely used in the field of genomics [18]. Lorién et al. show that the SIMD version of chain obtains slowdowns of up to 0.71× with respect to the scalar version with heuristics [37]. Chain presents dependency-bound patterns in the inner loop, resulting in underutilized vector lanes. Similarly, Guo et al. show that in the GPU version of chain, 16.3% of the kernel instructions are control instructions for synchronizing warps [33]. They use an NVIDIA Tesla P100 for the evaluation [38]. The P100 GPU achieves a 3.17× speed-up with respect to a 14-core CPU while consuming 300W and occupying 610 mm^2, resulting in under-utilization of resources.

These limitations highlight the need for a flexible and efficient solution to exploit fine-grain parallelism in dependency-bound workloads. We propose a general-purpose accelerator - Squire - to unlock the parallelism potential of a wide range of workloads while maintaining flexibility and low overhead.

III. Use cases

In this section, we discuss three potential use cases for our accelerator: data sorting, genomics, and signal processing. We examine some representative kernels and demonstrate how fine-grain parallelism can be exploited.

A. Data Sorting

Sorting [17] is a widely studied problem in computer science, fundamental to various applications such as search engines, data mining, databases, and numerical methods. The importance of sorting spans from energy-efficient devices [39], [40] to GPGPUs and data centers [41]–[43].

Radix sort [17] is an efficient sorting algorithm with a time complexity of $O(nk)$, where n is the number of elements in the array and k is the length of the key. This makes it particularly effective to sort arrays of 32-bit or 64-bit integers. In the first iteration, the algorithm uses the eight most significant bits to divide the array into 2^8 buckets. This process is repeated recursively for subsequent bits until all bits are processed.

Fine-grain parallelism in Radix Sort can be achieved by dividing the array into smaller chunks, sorting them independently, and merging the results. Previous work has shown how GPGPUs can achieve this using parallel architectures [44]. We will show that Squire can also leverage this parallelism while experiencing minimal synchronization overhead.

B. Genomics

In life science research and health care, sequencing technologies have revolutionized the way scientists analyze the genome to uncover biological insights. Modern sequencing technologies can accurately read massive amounts of genome sequences. Afterwards, tools like read mappers are extensively used in multiple sequence data analysis methods to locate (align) the read sequences in a reference genome (e.g., the human genome). Since alignment algorithms are computationally expensive, read mappers usually follow a seed-and-extend approach. During the seeding stage, the tool searches for partial matches between the query sequence and the reference. These partial matches are hints for the extend stage, where dynamic programming algorithms find the best location for each sequence.

The goal of **seeding** is to find partial exact matches between an input sequence and the reference genome. Typically, FM-Index [28] or hash-tables [17] are used to locate these matches faster. These data structures are built once and used along all the sequences being aligned, typically several GBs or even TBs of data.

Minimap2 [18] is a well-known read-mapper tool used for long reads (around 10K base pairs). Initially, Minimap2 builds a hash table that contains the positions in the reference for all combinations of k base pairs (k-mer). When Minimap2 indexes the hash table with a k-mer, the hash table returns a list of positions in the reference. To split the sequence into k-mers, Minimap2 establishes a sliding window that moves along the sequence and extracts the lowest k-mer alphabetically in each window. Each one of these k-mers is called a minimizer.

Minimap2 indexes the hash table with all the minimizers and extracts a list of tuples (called anchors), which consist of the position in the sequence and the position in the reference. Finally, the anchors are sorted by the position in the reference so that the following stages can easily traverse the list. For this purpose, Minimap2 uses a radix sort algorithm, which is the most time-consuming step of the entire seeding stage.

The Chain kernel, a 1D dynamic programming algorithm, combines multiple seeds (termed anchors) to create extended matching regions, also called a *chains*. The algorithm receives a set of sorted anchors and scores each anchor pair based on their proximity and overlap using the following formula:

$$f(i) = \max_{(i-T)\leq j<i} \{f(j) + \alpha(i,j) - \beta(i,j)\} \qquad (1)$$

where $f(i)$ is the score of anchor i, $\alpha(i,j)$ is a bonus score between the anchors i and j, and $\beta(i,j)$ a penalty for gaps and overlaps. Finally, T is the chain iteration threshold. Notice that the $f(i)$ calculation depends on $f(i-1)$. On the other hand, the calculation of α and β is independent for any i and j.

Figure 2 shows the dependencies present in the chain kernel. The score array (f in Equation 1) is added to the match-up scores (α and β in Equation 1). Then, the maximum of the row is calculated, and the score of anchor 4 is obtained. Notice that $f(4)$ will be added to subsequent rows, thus creating a dependency in the outer loop.

In addition to the score array, a predecessor array is computed, which stores the index of the best match-up for each

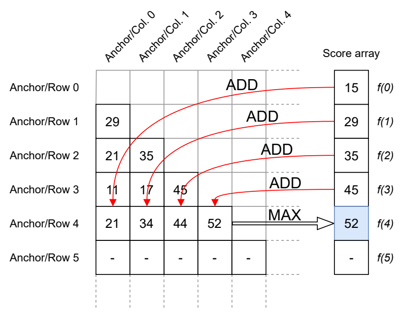

Fig. 2: Chain algorithm dependencies. To calculate the score of anchor 4 ($f(4)$), we add all the previous scores to row 4 and perform a maximum. In the next iteration, $f(4)$ will be added to row 5.

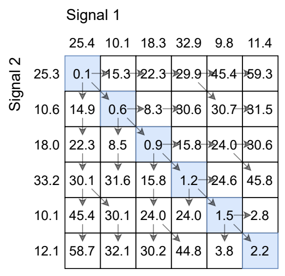

Fig. 3: DTW matrix representation. The samples from signal 1 are set at the top of the matrix, while samples from signal 2 are set at the left. For each cell DTW computes Equation 2. Arrows indicates the minimum value from Equation 2.

anchor. Once both arrays are computed, a backtracking process is performed. Starting from the best score in the score array, the algorithm recursively traces back through its predecessors. The resulting list of predecessors forms a *chain*.

Dynamic programming 1D algorithms typically involve an inner loop with dependencies across iterations, which limits the amount of parallelism that can be exploited. The fine-grain parallelism arises from the computation of elements that depend on previously computed iterations. As illustrated in Figure 2, the element $f(4)$ in the score array requires all elements smaller than 4. However, it is unnecessary to wait for $f(3)$ to complete before calculating certain cells in the row. For example, we can compute cell x in row 4 as soon as $f(x)$ is available by dynamically checking the computed scores. A system of detached computing elements could facilitate such computations effectively.

Smith-Waterman is a 2D dynamic programming algorithm used during the extend stage for sequence alignment [19], [20]. As this paper discusses another 2D dynamic programming algorithm, Dynamic Time Warping, in detail later, we do not elaborate further on Smith-Waterman here.

C. Signal Processing

Signal processing focuses on analyzing various types of signals, including sound, images, potential fields, seismic data, altimetry, and scientific measurements. A signal represents a flow of information originating from a source, which can take many forms, such as mechanical, optical, magnetic, electrical, or acoustic. Signals can be digital, characterized by discrete values, such as semaphores, Morse code, or the contents of computer memory. Conversely, signals can also be analog, encompassing continuous values like pressure, temperature, or velocity.

Dynamic Time Warping (DTW) is a 2D dynamic programming algorithm designed to align two signals to measure their similarity. It is widely used in applications such as speech recognition, speaker recognition, and music recognition. DTW constructs a dynamic programming matrix with a computational complexity of $O(n \times m)$, where n and m represent the lengths of the signals being aligned.

Figure 3 provides an example of a DTW matrix. Each cell in the matrix is computed using the following equation:

$$\begin{aligned} M[i,j] = abs(S[i] - R[j]) + \min\{&M[i-1,j-1], \\ &M[i-1,j], \\ &M[i,j-1]\} \end{aligned} \tag{2}$$

Where $M[i,j]$ is the cell in row i and column j, while S and R are the aligned signals. The equation calculates the value of the cell $[i,j]$ as the minimum value of the left, top, and left-top cells plus the absolute difference between $S[i]$ and $R[j]$. This dependency on the left, top, and left-top cells complicates parallelization.

Similarly, the aforementioned Smith-Waterman algorithm also exhibits the same dependency patterns as DTW. Like DTW, Smith-Waterman also constructs a matrix with dependencies on the left, top, and left-top cells.

Typically, fine-grain parallelism for 2D dynamic programming kernels is primarily achieved using SIMD techniques. The most common approaches are anti-diagonal vectorization [45] and inter-task vectorization [46]. Both methods require prior reorganization of the data structures and often suffer from under-utilization of SIMD lanes.

On one hand, SIMD enforces a lockstep execution order, limiting flexibility. On the other hand, dividing work in this manner can lead to load imbalance. A potential solution to these issues is to increase the size of the work chunks assigned to each computing element. As shown in Section V-C, the dynamic programming (DP) matrix could be divided by columns and distributed among the computing elements. To implement this effectively, we would require a set of independent computing elements capable of asynchronously processing their assigned chunks of work.

D. Discussion

We have demonstrated that fine-grain parallelism exists in several key dependency-bound kernels. Exploiting this parallelism effectively requires a set of generic, independent, general-purpose computing elements. Such a system must include a mechanism for rapid work offloading and efficient synchronization among the computing elements. Building on these principles, the next section introduces Squire, a general-purpose accelerator designed specifically to address the challenges of dependency-bound fine-grain parallelism.

IV. Squire

One of the main goals of Squire is to make it general-purpose, enabling workloads with inherent fine-grain parallelism to benefit from the accelerator. To achieve this, we identify two key design principles: (i) enable low-latency execution of fine-grain tasks and (ii) provide architectural support for fast synchronization between processing units to manage dependencies. Hence, our hardware accelerator must have the following features:

- A set of general-purpose processing units sharing a unified memory view with the host core.
- A synchronization mechanism to enable rapid communication among processing units.

A. Squire Design

Figure 4 shows the architectural overview of Squire, a general-purpose accelerator for dependency-bound fine-grain parallelism. Figure 4a illustrates a conventional multi-core SoC with a distributed L3 cache, where each core complex contains two levels of private caches. Figure 4b depicts the integration of Squire into the system, where each core complex is augmented with a Squire block interfaced with the private L2 cache. Finally, Figure 4c shows that Squire consists of a set of very simple general-purpose in-order cores, termed *workers*. In addition, Squire features control registers and a synchronization module, which are visible to both the host core and the workers.

Typically, hardware accelerators solve dependency-bound parallelism, such as dynamic programming, using systolic arrays [14], [33], [35]. However, these solutions rely on hard-wired components that serve a fixed purpose and usually target a specific kernel. In order to increase the flexibility of Squire, we propose employing simple in-order cores with small area and power consumption requirements for each worker. To simplify the design, we assume these cores share the same base ISA as the host core. In addition, each worker has small, private data and instruction caches. We define the size of these caches with a design space study in Section VII-D.

A host core can offload computation to the workers via a simple API (see Section IV-C) that sets a function's address and the necessary arguments into the control registers. Then, the workers start executing the workload using regular instructions. If the host core has recently accessed the input data, it is likely to still reside in the L2 cache, reducing data transfer latency.

To orchestrate L2 access requests from the worker cores, we employ a shared bus coupled with a centralized arbiter. The arbiter selects one request per cycle from the set of pending L2 accesses issued by the workers. This design enforces a single L2 access per cycle, thereby requiring only a single extra read/write port on the L2 cache, reducing implementation complexity. Cache coherence is maintained through a snoop-based protocol, where all workers monitor the L2 bus for invalidation messages. This is practical given the simplicity of the in-order cores. Moreover, workers are designed to target workloads that maximize L1 data reuse. Empirically, even with 32 workers active, the system sustains an average of no more than one L2 access every two cycles, demonstrating that accessing the L2 cache is not a primary bottleneck.

TABLE I: Squire programming interface. For each API call, we provide a brief description and who can use the API call.

API call	Description	Caller
`start_squire(f,a)`	Squire executes `f` function with `a` arguments. *Counters* reset to 0.	Core
`stop_worker()`	Suspends the worker execution.	Workers
`id_worker()`	Returns the worker ID.	Workers
`num_workers()`	Returns the total number of workers.	Core/Workers
`inc_lcounter(w)`	Increments the *local counter* `w` by one.	Workers
`inc_gcounter()`	Increments the *global counter* by one.	Workers
`wait_lcounter(w,s)`	Waits until the *local counter* `w` is greater or equal to `s`.	Core/Workers
`wait_gcounter(s)`	Waits until the *global counter* is greater or equal to `s`.	Core/Workers

Tasks are distributed using traditional coarse-grain parallelism, with OpenMP assigning independent workloads to host cores [47]. In read mapping tools, for example, each host core aligns a subset of sequences. These tasks are typically dependency-bound, limiting the effectiveness of SIMD and instruction-level parallelism. Squire addresses this by subdividing tasks into fine-grain sub-tasks, enabling nested parallelism even in dependency-bound kernels.

B. Synchronizing Workers

The synchronization mechanism is used to coordinate the workers, and it is visible to the host core as well as the workers. We have designed the mechanism to enable modeling dependencies for two distinct common use cases.

On the one hand, our aim is to tackle algorithms that perform computation over 1D data structures. To achieve this, we use a simple mechanism that features a hardware atomic counter, referred to as *global counter*. This will enable handling loops where iteration *i* conditionally consumes the data produced by iteration *i-1*. For this purpose, we require the workers to increment the *global counter* in order, i.e., if worker *x* increments the *global counter* before worker *x-1*, the increment is saved in a structure until worker *x-1* increments the *global counter*. To implement this with a non-blocking scheme, we instantiate one queue per worker and a token. The token indicates which worker is the next to increment the *global counter* and is initialized to zero. If worker *x* wants to increment the *global counter* and the token contains the value *x-1*, an increment request is enqueued in *x*'s queue. When worker *x-1* increments the *global counter*, the queues are searched for pending increments in order, and the token is updated accordingly.

On the other hand, we want to efficiently handle workloads with 2D data structures, such as dynamic programming matrices with vertical and horizontal dependencies. For this reason, we also instantiate an array of hardware atomic counters, with a length equal to the number of workers, referred to as

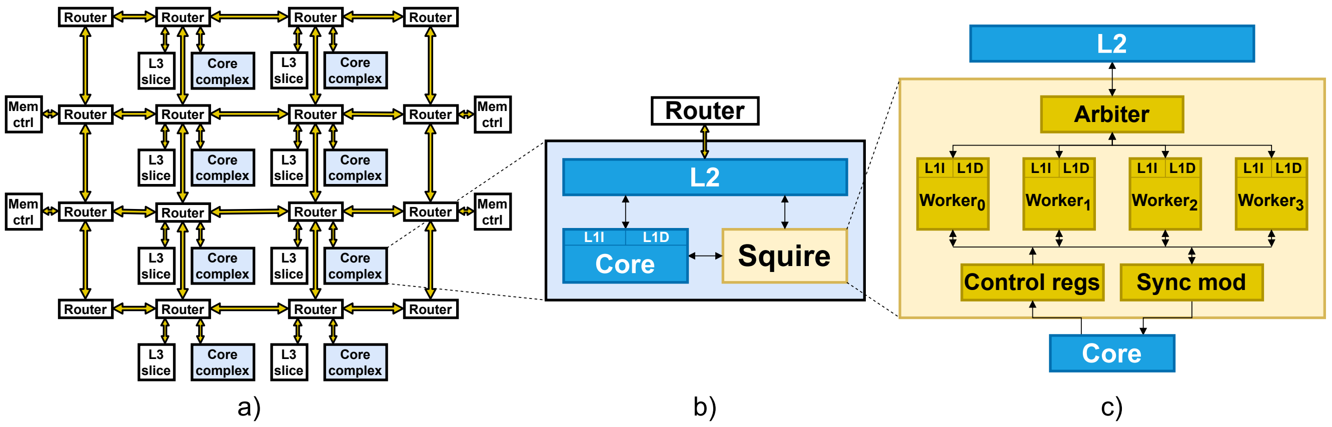

Fig. 4: Squire architectural overview: (a) the simulated multi-core system with a 4x4 NoC and four memory controllers, where each central router holds a core complex and one slice of the L3 cache; (b) a core complex contains one OoO core with private L1 caches, a private L2 cache, and a Squire; (c) Squire contains a set of workers, several control registers, a synchronization module, and an arbiter to communicate with the memory hierarchy (L2).

local counters. When filling a dynamic programming matrix, a worker increments its *local counter* each time it computes a matrix row. Thus, worker *x* can check *local counter x-1* before starting the next row.

This set of hardware atomic counters is implemented as 64-bit registers that can be accessed in one cycle.

C. Squire API

Table I describes the Squire programming interface. Each functionality in the table defines a new ISA primitive to interact with the accelerator. The table also specifies whether the host core, the workers, or both can invoke the primitives. Section V-A shows how the Squire API works using radix sort as an example.

V. Using Squire

This section shows how to use Squire for several kernels. We describe the implementation process for the Radix Sort, Chain, and DTW algorithms in Squire. Finally, we discuss some alternatives considered for certain implementation details.

A. Sorting: Radix Sort

The pseudocode for Squire's radix sort implementation is shown in Algorithm 1. The host core executes the `RADIX` function (Line 1), which calls `start_squire` with the function and input data addresses as arguments (Line 3). This call writes the addresses to Squire's control registers, sets the workers' program counters to the function's entry point, and resets internal counters. The workers then execute the `RADIX_Workers` function (Line 8). Each worker retrieves its ID and the total number of workers via the `id_worker` and `num_workers` APIs, using this information to evenly partition the input array (Lines 9–10). Each chunk is sorted using the standard radix sort algorithm (Line 11). Upon completing its chunk, a worker increments the global counter and halts (Lines 12–13). Meanwhile, the host core waits until

Algorithm 1 Radix Sort Squire version

```
1:  function RADIX(X[N])
2:      if N > 10000 then
3:          start_squire(RADIX_WORKERS, X)
4:          wait_gcounter(num_workers())
5:          MERGE_SORTED_ARRAYS(X)
6:      else
7:          RADIX_KERNEL(X[0:N])
8:  function RADIX_WORKERS(X[N])
9:      start = id_worker() × (N / num_workers())
10:     end = (id_worker()+1) × (N / num_workers())
11:     RADIX_KERNEL(X[start:end])
12:     inc_gcounter()
13:     stop_worker()
```

the global counter matches the number of workers (Line 4). At this point, the input has been divided into n sorted subarrays, which the host core merges using a min-heap (Line 5). Squire may not be beneficial when the workload is too small. To address this, Algorithm 1 includes a check to ensure that at least 10,000 elements are present before activating Squire (Line 2); otherwise, the host core handles sorting directly (Line 7).

B. 1D Dynamic Programming: Chain

We describe the process of integrating the Chain kernel into Squire. First, we show the pseudocode for the baseline version of the Chain kernel. Next, we outline generic software modifications to enable parallelism, and finally, we show the necessary changes to integrate Chain into Squire.

1) Baseline Chain Kernel

Algorithm 2 shows the pseudocode for the original chain kernel. The function `CHAIN` receives an array of anchors sorted by position in the reference (Line 1). The kernel consists of two nested loops. The outer loop (Line 3) goes through

Algorithm 2 Chain kernel baseline version

```
1: function CHAIN(A[N])                         ▷ A: anchors array
2:     T = 5000
3:     for i = 0; i < N; i++ do
4:         for j = i-1; j ≥ i-T; j-- do
5:             AUX[j] = $\alpha$(A[i], A[j]) - $\beta$(A[i], A[j])
6:             AUX[j] += F[j]                    ▷ Consume F[j]
7:         F[i] = MAX(AUX)                       ▷ Generate F[i]
```

all the anchors sorted by reference position, while the inner loop (Line 4) iterates through the `T` anchors prior to anchor `i` and performs a *match-up* between each of them and anchor `i`. Line 5 corresponds to the calculation of α and β in Equation 1, and Line 6 to the addition of $f(j)$ in Equation 1. Line 7 performs the maximum, obtaining $f(i)$. Notice that Line 5 can be computed in parallel for all the anchors, while Line 6 must wait for the generation of `F[j]` by Line 7.

2) *Enabling Fine-Grain Parallelism*

To delay the consumption of `F[i]` from Line 7 to Line 6, we alter the order of the inner loop (Line 4). To achieve this, we traverse the anchors in reverse order, i.e., from `i-T` to `i-1`. In addition, to isolate dependency-free parallelism from the dependencies imposed by Line 6, we fission the inner loop (Line 4), effectively detaching the computation of α and β in Line 5 from the addition in Line 6.

By default, the chain algorithm has a threshold on the number of anchors it visits backward (`T` in Lines 2 and 4). However, the best match-up is typically found during the initial iterations. In addition, the chain implements some heuristics to stop the inner loop earlier. For example, if the match-up scores are below a threshold. Therefore, the chain kernel visits fewer anchors, and typically only the first few are useful. Consequently, we can reduce `T` with a negligible penalization in accuracy. We observe a misprediction rate lower than 9 per million when setting `T` to 64. Therefore, we use this value for our final evaluation. Although the overall Minimap2 accuracy remains almost unchanged, limiting `T` skips some match-ups, and some computation shifts to the align stage.

In the original implementation, after performing the chain stage, there is a second opportunity to rerun the chain algorithm if the area covered by the anchors is not large enough. The chain kernel is executed again with looser parameters and simpler versions of α and β functions. We modify the second chain run to use the same function as in the first chain run while applying the new parameters. This simplifies the implementation process while preserving the output of the original algorithm. As a summary of the software modifications:

- We have reversed the order in which we traverse the inner loop (Line 4).
- We fission the inner loop. Line 5 will be executed in the first loop, and Line 6 in the second one.
- We limit the number of anchors visited backward to 64 (`T` in Lines 2 and 4).
- We reformulate the second chain run to use the same function as in the first run.

Algorithm 3 Chain kernel Squire version

```
1: function CHAIN_WORKERS(A[N])  ▷ A: Anchors array
2:     T = 64
3:     for i = id_worker(); i < N; i += num_workers() do
4:         for j = i-T; j ≤ i-1; j++ do
5:             AUX[j] = $\alpha$(A[i], A[j]) - $\beta$(A[i], A[j])
6:         for j = i-T; j ≤ i-1; j++ do
7:             if AUX[j] ≠ -$\infty$ then
8:                 wait_gcounter(j+1)
9:                 AUX[j] += F[j]                ▷ Consume F[j]
10:        F[i] = MAX(AUX)                       ▷ Generate F[i]
11:        inc_gcounter()
12:    stop_worker()
```

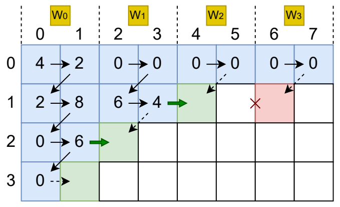

Fig. 5: DTW work distribution among workers. Worker 0 (W_0) computes columns 0 and 1, worker 1 (W_1) columns 2 and 3, worker 2 (W_2) columns 4 and 5, and worker 3 (W_3) columns 6 and 7. The workers compute the cells following the path indicated by the arrows.

Algorithm 3 shows the modified code with all these changes.

3) *Squire Integration*

Algorithm 3 shows the modified chain kernel adapted for Squire. The work is divided in a round-robin fashion (Line 3); e.g., with four workers, worker 0 computes the scores of anchors (0, 4, 8, ...), worker 1 computes the scores of anchors (1, 5, 9, ...), and so on.

Note that now the loop in Line 4 can be computed in parallel without dependencies using the workers. Once all the α and β values have been computed, the second loop in Line 6 proceeds to compute the remaining part, which has dependencies across workers (red lines in Figure 2). The dependencies are expressed by waiting on the *global counter* until it contains the desired value (Line 8), which means the dependent `F[j]` has been computed. Once the current `F[i]` is computed (Line 10), we increment the *global counter* to notify (Line 11) the consumers of that value.

When β (the penalization score) is high enough, we can stop the computation for that match-up. For these cases, we add a conditional statement (Line 7). Note that bypassing the `wait_gcounter` instruction could cause a race condition. For this purpose, we have implemented the mechanism described in Section IV-B, where we enforce the order of the increments in the *global counter*.

Algorithm 4 DTW kernel Squire version

```
1:  function DTW_WORKERS(A[N], B[M])
2:      start = id_worker() × (M / num_workers())
3:      end = (id_worker() + 1) × (M / num_workers())
4:      for i = 0; i < N; i++ do
5:          if id_worker() ≠ 0 then
6:              wait_lcounter(id_worker()-1, i+1)
7:          for j = start; j < end; j++ do
8:              PREV ← MIN(M[i-1,j], M[i,j-1], M[i-1,j-1])
9:              COST ← COST_FUNC(A[i], B[j])
10:             M[i,j] ← PREV + COST
11:         inc_lcounter(id_worker())
12:     stop_worker()
```

C. 2D Dynamic Programming: DTW

We now detail how to use Squire to exploit fine-grain parallelism in DTW. Other well-known 2D DP kernels (e.g., Smith-Waterman, Needleman-Wunsch, etc.) exhibit the same patterns when computing the DP matrix.

Figure 5 shows a graphical scheme of how Squire would compute the DTW matrix. A set of consecutive columns is assigned to each worker; worker 0 (W_0) computes columns 0 and 1, worker 1 (W_1) columns 2 and 3, and so on. Each cell *(i,j)* of the matrix has a dependency with cells *(i-1,j)*, *(i,j-1)* and *(i-1,j-1)*. The workers compute their columns in a row-wise order. Hence, they do not have to worry about the vertical and diagonal dependencies. To solve horizontal dependencies at the boundaries, the *local counters* from the *synchronization module* are used.

Algorithm 4 shows the pseudocode for the Squire version of DTW. First, the work is evenly divided among the workers (Lines 2 and 3). The outer loop iterates through the rows (Line 4), while the inner loop iterates through the assigned columns of the corresponding worker (Line 7). The equations of DTW are implemented in Lines 8, 9, and 10. To synchronize workers at the boundaries, worker *x* increments the *local counter x* when it finishes a row (Line 11), so worker *x+1* knows the dependency for that row is solved. Similarly, when worker *x* starts a row, it waits for worker *x-1* to finish its chunk of the row (Line 6). Note that worker 0 has no horizontal dependencies. Therefore, it skips the synchronization (Line 5).

D. Discussion

Throughout the development of Squire, we have examined several ideas regarding certain implementation details.

For communication among the workers, we have considered message-passing through a crossbar, a FIFO, or a ring. Finally, we have used the shared L2 cache since the worker's messages are part of the output, avoiding the need to write the same data twice.

We also considered other synchronization mechanisms besides the *counters*. Initially, the message-passing mechanism would be used as the synchronization point. We explored expanding the *synchronization module* functionality, allowing subtractions and arbitrary additions over the counter. The current *synchronization module* specifications are sufficient for the algorithms we use, but they could be extended in the future.

Finally, as we explained in Section IV-A, workers must increase the *global counter* in order. We considered solving this problem in software by waiting for the *global counter* to reach its correct value before incrementing it, e.g., in Algorithm 3 adding `wait_gcounter(i)` between Lines 10 and 11. However, this approach would harm available parallelism and performance.

TABLE II: Simulated architectural parameters.

Cores	8 Neoverse-N1-like Armv8 out-of-order cores 2.4 GHz
Structure entries	ROB: 224 | LD/ST queues: 96/96 | Inst. queue: 120
OoO Private L1 I&D	64 KB, 4-way, 1 cycle data access, 32 MSHRs
Private L2	512 KB, 8-way, 4 cycle data access, 64 MSHRs
Shared L3	Mostly exclusive, 8 slices of 1 MB, 16-way, 10 cycles data access, 128 MSHRs
Coherence protocol	MOESI-like AMBA 5 CHI specification
Network topology	4×4 2D mesh, 1 cycle routers, 1 cycle links (Fig. 4a)
Memory	1 HBM2 stack, 300 GB/s
Worker	Cortex-M35P-like Armv8 4-stage dual-issue in-order cores 2.4 GHz

TABLE III: Size of the datasets used in the evaluation.

	RADIX	SEED	CHAIN	SW	DTW
# experiments	15	5	5	5	2
# inputs/exp.	8 arrays	24 seq.	24 arrays	6195 align.	5000 align.
Input avg. size	53536 elems.	23014 bps	53536 anchors	1373 bps	221 samples
Size st. dev.	36886	15075	36886	2950	101
Mem. footprint	837 KB	22.5 KB	837 KB	3.27 KB	1.72 KB

VI. Evaluation Methodology

A. Architectural Simulation

We prototype Squire using the gem5 simulator v23.0 [48], [49]. We simulate a multicore system consisting of 8 Neoverse-N1-like out-of-order cores, three levels of cache, 4 HBM2e memory channels, and a mesh-based network-on-chip modeling the AMBA 5 CHI protocol, as shown in Figure 4a. Each host core features a Squire engine that faithfully models the described architecture. The simulated system runs Ubuntu 22.04 with Linux kernel 5.4.65. Table II summarizes the architectural parameters.

B. Workloads and Inputs

Table III details the inputs used for each kernel. All the inputs have been extracted from real genomics and signal-processing datasets. To evaluate Squire, we use the five kernels described below.

Radix Sort (RADIX) Radix sort is shown in Section III-A. We did 15 experiments. In each experiment, we sort eight arrays, one for each out-of-order core. Some of the arrays used for radix sort have less than 10,000 elements, thus avoiding offloading work to Squire (see Section V-A). We divide the array into equal chunks and use Squire to sort them (see Section V-A).

Seeding (SEED) We evaluate the seeding algorithm from Minimap2 [18] (see Section III-B). We use five input sequences datasets (see Table IV). Each one of the datasets has 24 sequences, hence, each out-of-order core performs three seeding processes. The most consuming part of seeding is the final sorting of the seeds. Therefore, we use the Squire version of the radix sort algorithm explained above.

Chain (CHAIN) Chain is a dynamic algorithm used in Minimap2 [18] (see Section III-B). As in seeding kernel, we use five input sequences datasets, where each one has 24 sequences, resulting in three chain processes per out-of-order. The anchors are assigned to the workers in a round robin manner (see Section V-B).

Smith-Waterman (SW) Smith-Waterman is a 2D dynamic programming algorithm used for aligning (see Section III-B). We use the same datasets used in seeding and chain. These datasets produce several alignments that we use as inputs in Smith-Waterman. The work has been distributed using the same approach as for DTW (see Section V-C).

Dynamic Time Warping (DTW) Dynamic Time Warping is a 2D dynamic programming algorithm (explained in Section III-C) used for signal processing. We use two synthetic datasets of 5,000 alignments of floating point numbers. The small dataset has an average alignment size of 133 samples, while the larger one has 380 samples on average. Each worker is the responsible of computing several contiguous columns (see Section V-C).

C. Evaluation of an End-to-End Read-Mapping Application

With the kernels introduced above, we have built an end-to-end read-mapping tool that receives a set of sequences and produces alignments. We use Minimap2 [18] as the skeleton for our read-mapper since two of the evaluated kernels are extracted from Minimap2 (SEED and CHAIN). We combine SEED, CHAIN, and SW into a single application to set up a read-mapper that serves as a test-bench for evaluating the speed-up achieved on an end-to-end application when using Squire.

Table IV shows the inputs used to evaluate the end-to-end application. All these inputs are from sequencing machines that have sequenced the human genome. Note the differences in the accuracy of the inputs, which refer to the errors introduced by the machines during the sequencing (reading) process. ONT and PBCLR have an accuracy of 85% and 88%, respectively, while PBHF inputs have an accuracy of nearly 100%. PBHF inputs are obtained using “PacBio High Fidelity” technology, which consists of reading the same piece of genome several times and mitigating the error by a consensus process. This difference in accuracy is translated into different behavior during the read-mapping process. A higher accuracy implies a lighter volume of work in the align stage when using SW.

We select the 18 most time-consuming sequences from each input set to keep simulation time in gem5 manageable. This allows us to reduce the execution time while maintaining the application’s behavior.

TABLE IV: Input sequence datasets.

	Sequencing machine	Avg. seq. length	Accuracy
ONT [50]	Oxford Nanopore	17,710	85%
PB CLR [51]	PB Sequel II System	6,739	88%
PB HF 1 [50]	PacBio HiFi	12,858	99.99%
PB HF 2 [50]	PacBio HiFi	15,602	99.99%
PB HF 3 [50]	PacBio HiFi	14,149	99.99%

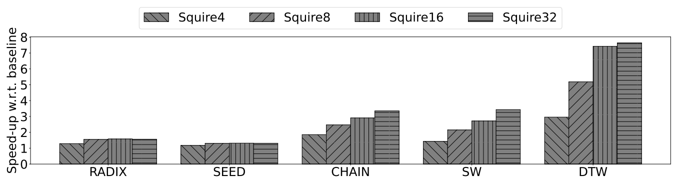

Fig. 6: Squire evaluation for the five kernels described in Section VI-B. We evaluate Squire with 4, 8, 16, and 32 workers.

VII. Evaluation

In this section, first, we show how Squire can speed up the five evaluated kernels. Then, we evaluate the impact the synchronization module has on the design by modifying the implementation to use software mutexes instead of Squire’s hardware module. We also evaluate the end-to-end read-mapper to understand how Squire improves a full application. Finally, we perform a design space exploration to justify the size of the caches used by the workers and perform an area and energy consumption study.

A. Performance Evaluation

Figure 6 shows the performance evaluation of Squire for the five kernels described in Section VI-B when changing the number of workers.

For RADIX and SEED, Squire achieves diminishing returns when using from 8 up to 32 workers, due to small input data size. As explained in Section V-A, we stablish a minimum of 10,000 elements to use Squire. Below that, the initialization of Squire becomes the bottleneck in the sorting process. Maximum performance is achieved with 16 workers, reaching 1.58× for RADIX and 1.32× for SEED.

Employing 32 workers for CHAIN and SW leads to noticeable speed-ups, unlike RADIX and SEED, reaching 3.35× and 3.43× with respect to the base system, respectively. The speedups from 16 to 32 workers are 1.19× and 1.26× for CHAIN and SW.

Finally, Squire obtains remarkable speedups up to 32 workers for DTW, reaching 7.64×. However, we consider 16 workers the optimum point with a speedup of 7.42×.

These results show that Squire can enable fine-grain parallelism on dependency-bond kernels. While Squire scales well with worker count if there is enough work to compute, we advocate that a balanced design should have between 8 and 16 workers. Doubling the number of workers to 32 does not compensate for the cost in the common case.

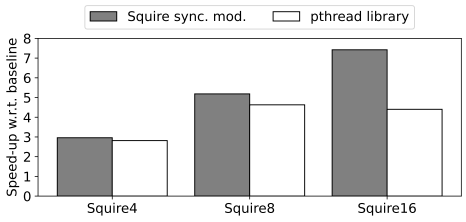

Fig. 7: Squire performance evaluation when using the synchronization module vs the pthread library.

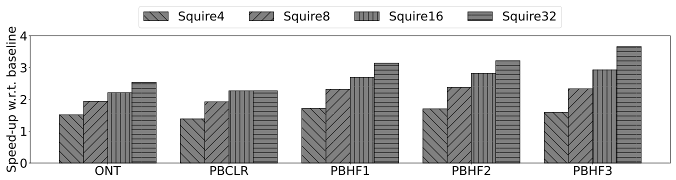

Fig. 8: Squire evaluation for the end-to-end read-mapping application described in Section VI-C.

B. Synchronization Module Evaluation

Figure 7 shows the benefits of using the synchronization module in Squire for DTW kernel. We show the results up to 16 workers since we have considered it the optimum point in Section VII-A. We instantiate Squire without the synchronization module and synchronize through the *pthread mutex* library. We use DTW for this experiment since it is one of the kernels (along with SW) that uses the *local counters*.

The synchronization module improves performance for any number of workers, increasing in importance as the number of workers increases. We observe a speed-up of up 1.69× when using the synchronization module with 16 workers.

C. End-to-End Application Evaluation

Figure 8 shows the performance evaluation of Squire for an end-to-end application for the five inputs described in Table IV. We evaluate Squire with 4, 8, 16, and 32 workers. As stated in Section VI-C, the different datasets behave differently during the read mapping process; thus, the align stage has less weight for the PBHF inputs.

When looking at the whole read-mapping end-to-end application, Squire achieves speed-ups of up to 3.66×. For all the inputs, Squire scales well with worker count and accomplishes its best performance with 32 workers. For ONT and PBCLR inputs, Squire achieves speed-ups of 2.54× and 2.27×, respectively. For PBHF inputs, Squire achieves speed-ups higher than 3×. A higher accuracy of the sequencing machines implies more work to process but smaller chunks of work, which favors Squire. As sequencing technologies keep improving, this trend will consolidate and devices like Squire will be more effective.

D. Cache Size Exploration

Each worker has its own private L1 data and instruction caches, which will largely determine the area that Squire will

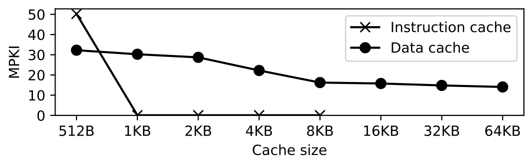

Fig. 9: Misses per kilo instructions (MPKI) when changing the cache size for the instruction and data caches.

occupy. For this reason, we perform a design space exploration study to make a judicious choice and minimize the area and power overhead of the design.

To evaluate the cache sizes, we use the end-to-end application and fix the number of workers to 16. We use the ONT input dataset. To evaluate the instruction cache size, we fixed the data cache size to 8 KB and vice versa. To measure performance, we use misses per kilo instructions (MPKI).

Figure 9 shows MPKI when varying cache sizes. For the instruction cache, we observe a drastic change when going from 512 B to 1 KB. Beyond that, MPKI remains close to zero. For the data cache, we see consistent improvement up to 8 KB, which we consider the sweet spot. A larger 16 KB data cache improves MPKI marginally at a large cost. Therefore, we have employed 1 KB and 8 KB as instruction and data cache sizes, respectively, for all the experiments in this section.

E. Area Overhead

We use the Arm Neoverse N1 to model the out-of-order core. Using the public data for an N1 [52] at 7nm, the floor planned area is given as 1.15 mm^2.

The workers we model could be compared to the Arm Cortex M35P microprocessor. Using public data for an M35P at 40LP [53], the floor planned area is given as 0.091 mm^2. This area already includes a 16 KB instruction cache. The instruction cache included in the M35P is larger than the caches we employ since we employ 1 KB for L1I and 8 KB for L1D (see Section VII-D). Also, the M35P is a processor capable of booting an operating system, and our workers do not require as many functionalities as the M35P. Therefore, we must consider that we are overestimating the area of the workers.

When employing 16 workers, the total area overhead at 40nm would be 1.456 mm^2. To estimate the area with 7 nm, we scale these numbers, considering fin pitch, gate pitch, and interconnect pitch, using data from several studies [54]–[59] to arrive at a 12× area reduction when moving from 40 nm to 7 nm. Thus, obtaining an area for a Squire component of 0.121 mm^2.

Therefore, we could place a 16-worker Squire component per core with an area overhead of 10.5%.

F. Energy Consumption

To estimate the Squire energy consumption, we use McPAT 1.3 [60] with the enhancements proposed by Xi et al. [61]. We performed this estimation using a process technology node

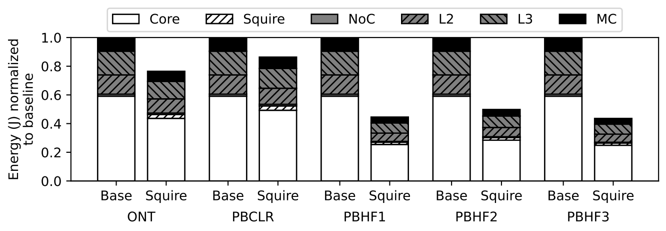

Fig. 10: Energy consumption comparison between the baseline and when using Squire for the end-to-end application.

of 22 nm, a supply voltage of 0.8 V, and the default clock gating scheme.

Figure 10 shows the energy consumption of the baseline when using Squire with 16 workers for the end-to-end application. The executions using Squire achieve significant energy reductions of up to 56% over the baseline system for the PBHF3 input. Similarly, Squire reduces consumption by 55% and 50% for PBHF1 and PBHF2 inputs. The ONT and PBCLR inputs show a more modest energy reduction of 24% and 14%, respectively.

The host cores are the most energy-consuming components, followed by the L2 and L3 caches. The memory controllers and the NoC have a marginal energy consumption. The energy overhead introduced by Squire is small; we observe an energy overhead of around 6% with respect to the host cores, which is largely offset by the reduction in the rest of the components.

VIII. Related Work

We identify more general-purpose hardware accelerators like the Walkers [62], a programmable hardware accelerator for traversing hash tables in a database. Transmuter [63] and Versa [64] propose a matrix of general-purpose processing elements interconnected by a mesh. The accelerator is shared by all the cores of the chip. The system can be reconfigured as a systolic array of processing elements, as a typical memory hierarchy, or as a private scratchpad for each processing element. UPMEM [65] is the first publicly available general-purpose programmable PIM system. AIM [66] is a sequence alignment framework that uses UPMEM for the evaluation.

Table V shows a qualitative comparison between Squire and the other general-purpose hardware accelerators. The Walkers have a fixed pipeline, forcing the data to traverse it, thus limiting its flexibility. In addition, the Walkers have a very limited ISA support and do not have any method for synchronization. The compute units must execute the code completely in parallel without communicating with the rest. Transmuter and Versa instantiate one accelerator shared for all the cores of a chip. To exploit all the computing resources, the application should be split into two sets of threads, one that is executed on the accelerator and the other on the cores. By contrast, Squire is a simpler private accelerator for each core. The application is divided into as many threads as cores, and then each core performs nested parallelism in its Squire. Moreover, the interconnection networks in Transmuter and

TABLE V: Qualitative comparison among several proposals.

	Walkers [62]	Transmuter [63]	Versa [64]	AIM [66]	Squire
Programable	✓	✓	✓	✓	✓
Rich ISA support	×	✓	✓	✓	✓
Flexible datapath	×	✓	✓	✓	✓
Virtual memory support	✓	✓	✓	×	✓
Rapid synchronization	×	✓	✓	×	✓
Private accelerator per core	✓	×	×	×	✓

Versa (among processing elements and between the accelerator and the host cores) add communication latencies with respect to Squire. AIM is a processing in memory component that instantiates several processors per physical memory cell, thus losing the virtual memory capability and limiting the address range the processors can access. Each processor controls a chunk of the memory and can not access the rest of the system. To communicate with other processors, they must do it through main memory, which hinders performance [67].

A big.LITTLE architecture is composed of several cores with different design targets: computational performance and power efficiency [68]. All are capable of running system code and are visible to the operating system. In contrast, Squire is a set of very simple cores with no system support and is subordinated to a host core. Moreover, Squire is equipped with a synchronization module that allows for fast communication among its workers (see Section VII-B).

IX. Conclusions

In this article, we propose Squire, a general-purpose accelerator for dependency-bound fine-grain parallelism. Squire consists of a set of simple general-purpose in-order cores, called workers, and a synchronization module for rapid synchronization. Each host core is augmented with a Squire engine to offload fine-grain tasks.

We evaluate Squire on a simulated multicore SoC, obtaining speed-ups of up to 7.64× in dynamic programming kernels, and an acceleration for an end-to-end application of 3.66×. We also evaluate the usage of resources and show that Squire achieves an energy reduction of up to 56% with an area overhead of 10.5% per core.

Acknowledgment

This research was supported by MICIU/AEI/10.13039/501100011033 and by "ERDF A way of making Europe" through contracts [PID2023-146511NB-I00], [PID2023-146193OB-I00], and [PID2022-136454NB-C22]; by the Ministry for Digital Transformation and Public Service, (i) via the framework of the Recovery, Transformation and Resilience Plan - NextGenerationEU [REGAGE22e00058408992] and (ii) through the Càtedra Chip UPC project [grant number TSI-069100-2023-0015]; by the Generalitat of Catalunya through contract [2021-SGR-00763]; by the Arm-BSC Center of Excellence; by the Lenovo-BSC Framework Contract (2020); and by Government of Aragón (T58_23R research group). A. Armejach is a Serra Hunter Fellow.

References

[1] M. Dayarathna, Y. Wen, and R. Fan, "Data center energy consumption modeling: A survey," *IEEE Communications surveys & tutorials*, vol. 18, no. 1, pp. 732–794, 2015.

[2] Y. Hu, Y. Liu, and Z. Liu, "A Survey on Convolutional Neural Network Accelerators: GPU, FPGA and ASIC," in *2022 14th International Conference on Computer Research and Development (ICCRD)*. IEEE, Jan. 2022. [Online]. Available: http://dx.doi.org/10.1109/ICCRD54409.2022.9730377

[3] A. Conesa, P. Madrigal, S. Tarazona, D. Gomez-Cabrero, A. Cervera, A. McPherson, M. W. Szcześniak, D. J. Gaffney, L. L. Elo, X. Zhang, and A. Mortazavi, "A survey of best practices for rna-seq data analysis," *Genome Biology*, vol. 17, no. 1, Jan. 2016. [Online]. Available: http://dx.doi.org/10.1186/s13059-016-0881-8

[4] R. Stallman and R. McGrath, *GNU make: A program for directing recompilation : GNU make version 3.79.1*. Free Software Foundation, 2002.

[5] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen, "LAPACK: A portable linear algebra library for high-performance computers," in *Proceedings SUPERCOMPUTING'90*. IEEE Computer Society, 1990, pp. 2–11.

[6] D. Lustig and M. Martonosi, "Reducing GPU offload latency via fine-grained CPU-GPU synchronization," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2013, pp. 354–365.

[7] S. Che, G. Rodgers, B. Beckmann, and S. Reinhardt, "Graph coloring on the gpu and some techniques to improve load imbalance," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 2015, pp. 610–617.

[8] B. Plancher and S. Kuindersma, "A performance analysis of parallel differential dynamic programming on a gpu," in *Algorithmic Foundations of Robotics XIII: Proceedings of the 13th Workshop on the Algorithmic Foundations of Robotics 13*. Springer, 2020, pp. 656–672.

[9] P. Steffen, R. Giegerich, and M. Giraud, "GPU parallelization of algebraic dynamic programming," in *Parallel Processing and Applied Mathematics: 8th International Conference, PPAM 2009, Wroclaw, Poland, September 13-16, 2009, Revised Selected Papers, Part II 8*. Springer, 2010, pp. 290–299.

[10] V. Boyer, D. El Baz, and M. Elkihel, "Dense dynamic programming on multi GPU," in *2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*. IEEE, 2011, pp. 545–551.

[11] K.-E. Berger and F. Galea, "An efficient parallelization strategy for dynamic programming on gpu," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 2013, pp. 1797–1806.

[12] D. Habich, J. Pietrzyk, A. Krause, J. Hildebrandt, and W. Lehner, "To use or not to use the SIMD gather instruction?" in *Proceedings of the 18th International Workshop on Data Management on New Hardware*, 2022, pp. 1–5.

[13] K.-C. Wu and Y.-W. Tsai, "Structured ASIC, evolution or revolution?" in *Proceedings of the 2004 international symposium on Physical design*, 2004, pp. 103–106.

[14] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly," in *ACM SIGPLAN Notices*, vol. 53, no. 2. ACM, 2018, pp. 199–213.

[15] Y. Zhang, X. Liao, H. Jin, L. He, B. He, H. Liu, and L. Gu, "DepGraph: A Dependency-Driven Accelerator for Efficient Iterative Graph Processing," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 371–384.

[16] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, and R. Narayanaswami, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017. [Online]. Available: http://dx.doi.org/10.1145/3079856.3080246

[17] D. Knuth, *The Art of Computer Programming*. Addison-Wesley, 1973, vol. 3.

[18] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 2018.

[19] M. S. Waterman, T. F. Smith, and W. A. Beyer, "Some biological sequence metrics," *Advances in Mathematics*, vol. 20, no. 3, pp. 367–387, 1976.

[20] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195 – 197, 1981.

[21] R. Bellman and R. Kalaba, "On adaptive control processes," *IRE Transactions on Automatic Control*, vol. 4, no. 2, pp. 1–9, 1959.

[22] M. Vasimuddin, S. Misra, H. Li, and S. Aluru, "Efficient architecture-aware acceleration of bwa-mem for multicore systems," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 314–324.

[23] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows–wheeler transform," *bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[24] ——, "Fast and accurate long-read alignment with burrows–wheeler transform," *Bioinformatics*, vol. 26, no. 5, pp. 589–595, 2010.

[25] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome," *Genome Biology*, vol. 10, no. 3, p. R25, 2009.

[26] S. Marco-Sola, M. Sammeth, R. Guigó, and P. Ribeca, "The gem mapper: fast, accurate and versatile alignment by filtration," *Nature methods*, vol. 9, no. 12, p. 1185, 2012.

[27] C. A. R. Hoare, "Algorithm 64: quicksort," *Communications of the ACM*, vol. 4, no. 7, p. 321, 1961.

[28] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proceedings 41st Annual Symposium on Foundations of Computer Science*. IEEE, 2000, pp. 390–398.

[29] J. M. Herruzo, S. G. Navarro, P. Ibáñez, V. Viñals-Yufera, J. Alastruey, and O. Plata, "Accelerating sequence alignments based on fm-index using the intel knl processor," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2018.

[30] R. Langarita, A. Armejach, J. Setoain, P. Ibáñez, J. Alastruey-Benedé, and M. M. Planas, "Compressed sparse fm-index: Fast sequence alignment using large k-steps," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2020.

[31] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," Nvidia Technical Report NVR-2008-004, Nvidia Corporation, Tech. Rep., 2008.

[32] NVIDIA, "Boosting Dynamic Programming Performance Using NVIDIA Hopper GPU DPX Instructions," https://developer.nvidia.com/blog/boosting-dynamic-programming-performance-using-nvidia-hopper-gpu-dpx-instructio 2022, accessed: 2024-07-15.

[33] L. Guo, J. Lau, Z. Ruan, P. Wei, and J. Cong, "Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between fpga and gpu," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 127–135.

[34] H. Sadasivan, M. Maric, E. Dawson, V. Iyer, J. Israeli, and S. Narayanasamy, "Accelerating Minimap2 for accurate long read alignment on GPUs," *bioRxiv*, 2022.

[35] D. S. Cali, G. S. Kalsi, Z. Bingol, C. Firtina, L. Subramanian, J. S. Kim, R. Ausavarungnirun, M. Alser, J. Gomez-Luna, A. Boroumand, A. Nori, A. Scibisz, S. Subramoney, C. Alkan, S. Ghose, and O. Mutlu, "GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020. [Online]. Available: http://dx.doi.org/10.1109/micro50266.2020.00081

[36] D. Fujiki, A. Subramaniyan, T. Zhang, Y. Zeng, R. Das, D. Blaauw, and S. Narayanasamy, "Genax: a genome sequencing accelerator," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018, pp. 69–82.

[37] L. López-Villellas, R. Langarita-Benítez, A. Badouh, V. Soria-Pardos, Q. Aguado-Puig, G. López-Paradís, M. Doblas, J. Setoain, C. Kim, M. Ono *et al.*, "Genarchbench: A genomics benchmark suite for arm hpc processors," *Future Generation Computer Systems*, vol. 157, pp. 313–329, 2024.

[38] NVIDIA, “NVIDIA Tesla P100,” https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016, accessed: 2024-07-31.

[39] M. Rashid, L. Ardito, and M. Torchiano, “Energy consumption analysis of algorithms implementations,” in *2015 ACM/IEEE International Symposium on Empirical software engineering and measurement (ESEM)*. IEEE, 2015, pp. 1–4.

[40] C. Bunse, H. Höpfner, S. Roychoudhury, and E. Mansour, “Choosing the” best” sorting algorithm for optimal energy consumption,” in *International Conference on Software and Data Technologies*, vol. 1. SCITEPRESS, 2009, pp. 199–206.

[41] D. P. Singh, I. Joshi, and J. Choudhary, “Survey of gpu based sorting algorithms,” *International Journal of Parallel Programming*, vol. 46, pp. 1017–1034, 2018.

[42] T. B. Chandra, V. Patle, and S. Kumar, “New horizon of energy efficiency in sorting algorithms: green computing,” in *Proceedings of National Conference on Recent Trends in Green Computing. School of Studies in Computer in Computer Science & IT, Pt. Ravishankar Shukla University, Raipur, India*, 2013, pp. 24–26.

[43] N. Schmitt, S. Kamthania, N. Rawtani, L. Mendoza, K.-D. Lange, and S. Kounev, “Energy-efficiency comparison of common sorting algorithms,” in *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2021, pp. 1–8.

[44] H. Nguyen, *GPU gems 3*. Addison-Wesley Professional, 2008, ch. 39.

[45] A. Wozniak, “Using video-oriented instructions to speed up sequence comparison,” *Bioinformatics*, vol. 13, no. 2, pp. 145–150, 1997.

[46] T. Rognes, “Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation,” *BMC bioinformatics*, vol. 12, no. 1, p. 221, 2011.

[47] R. Chandra, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[48] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011, Source: ACM SIGARCH Computer Architecture News ; volume 39, issue 2, page 1-7 ; ISSN 0163-5964. [Online]. Available: http://dx.doi.org/10.1145/2024716.2024718

[49] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillón, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, M. Fariborz, A. F. Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. S. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian, “The gem5 simulator: Version 20.0+,” *CoRR*, vol. abs/2007.03152, 2020. [Online]. Available: https://arxiv.org/abs/2007.03152

[50] precisionFDA, “PB HiFi/ONT,” https://precision.fda.gov/challenges/10, 2020, accessed: 2024-02-28.

[51] P. DevNet, “PB CLR,” https://github.com/PacificBiosciences/DevNet/wiki/HG002-Structural-Variant-Analysis-with-CLR-data, 2019, accessed: 2024-02-28.

[52] A. Pellegrini, N. Stephens, M. Bruce, Y. Ishii, J. Pusdesris, A. Raja, C. Abernathy, J. Koppanalil, T. Ringe, A. Tummala, J. Jalal, M. Werkheiser, and A. Kona, “The arm neoverse n1 platform: Building blocks for the next-gen cloud-to-edge infrastructure soc,” *IEEE Micro*, vol. 40, no. 2, pp. 53–62, 2020.

[53] Arm, “Arm Cortex-M35P,” https://developer.arm.com/Processors/Cortex-M35P, 2018, accessed: 2024-03-07.

[54] F. Arnaud, A. Thean, M. Eller, M. Lipinski, Y. Teh, M. Ostermayr, K. Kang, N. Kim, K. Ohuchi, J.-P. Han, D. Nair, J. Lian, S. Uchimura, S. Kohler, S. Miyaki, P. Ferreira, J.-H. Park, M. Hamaguchi, K. Miyashita, R. Augur, Q. Zhang, K. Strahrenberg, S. ElGhouli, J. Bonnouvrier, F. Matsuoka, R. Lindsay, J. Sudijono, F. Johnson, J. Ku, M. Sekine, A. Steegen, and R. Sampson, “Competitive and cost effective high-k based 28nm CMOS technology for low power applications,” in *2009 IEEE International Electron Devices Meeting (IEDM)*. IEEE, 2009. [Online]. Available: http://dx.doi.org/10.1109/iedm.2009.5424255

[55] K.-L. Cheng, C. C. Wu, Y. P. Wang, D. W. Lin, C. M. Chu, Y. Y. Tarng, S. Y. Lu, S. J. Yang, M. H. Hsieh, C. M. Liu, S. P. Fu, J. H. Chen, C. T. Lin, W. Y. Lien, H. Y. Huang, P. W. Wang, H. H. Lin, D. Y. Lee, M. J. Huang, C. F. Nieh, L. T. Lin, C. C. Chen, W. Chang, Y. H. Chiu, M. Y. Wang, C. H. Yeh, F. C. Chen, C. M. Wu, Y. H. Chang, S. C. Wang, H. C. Hsieh, M. D. Lei, K. Goto, H. J. Tao, M. Cao, H. C. Tuan, C. H. Diaz, and Y. J. Mii, “A highly scaled, high performance 45 nm bulk logic CMOS technology with 0.242 μm 2 SRAM cell,” in *2007 IEEE International Electron Devices Meeting*. IEEE, 2007. [Online]. Available: http://dx.doi.org/10.1109/iedm.2007.4418913

[56] D. C. Daly, L. C. Fujino, and K. C. Smith, “Through the looking glass-the 2018 edition: trends in solid-state circuits from the 65th isscc,” *IEEE Solid-State Circuits Magazine*, vol. 10, no. 1, pp. 30–46, 2018.

[57] S.-Y. Wu, C. Y. Lin, M. C. Chiang, J. J. Liaw, J. Y. Cheng, S. H. Yang, M. Liang, T. Miyashita, C. H. Tsai, B. C. Hsu, H. Y. Chen, T. Yamamoto, S. Y. Chang, V. S. Chang, C. H. Chang, J. H. Chen, H. F. Chen, K. C. Ting, Y. K. Wu, K. H. Pan, R. F. Tsui, C. H. Yao, P. R. Chang, H. M. Lien, T. L. Lee, H. M. Lee, W. Chang, T. Chang, R. Chen, M. Yeh, C. C. Chen, Y. H. Chiu, Y. H. Chen, H. C. Huang, Y. C. Lu, C. W. Chang, M. H. Tsai, C. C. Liu, K. S. Chen, C. C. Kuo, H. T. Lin, S. M. Jang, and Y. Ku, “A 16nm FinFET CMOS technology for mobile SoC and computing applications,” in *2013 IEEE International Electron Devices Meeting*. IEEE, 2013. [Online]. Available: http://dx.doi.org/10.1109/iedm.2013.6724591

[58] S.-Y. Wu, C. Lin, M. Chiang, J. Liaw, J. Cheng, S. Yang, C. Tsai, P. Chen, T. Miyashita, C. Chang, V. Chang, K. Pan, J. Chen, Y. Mor, K. Lai, C. Liang, H. Chen, S. Chang, C. Lin, C. Hsieh, R. Tsui, C. Yao, C. Chen, R. Chen, C. Lee, H. Lin, C. Chang, K. Chen, M. Tsai, K. Chen, Y. Ku, and S. M. Jang, “A 7nm CMOS platform technology featuring 4 th generation FinFET transistors with a 0.027um 2 high density 6-T SRAM cell for mobile SoC applications,” in *2016 IEEE International Electron Devices Meeting (IEDM)*. IEEE, 2016. [Online]. Available: http://dx.doi.org/10.1109/iedm.2016.7838333

[59] R. Xie, P. Montanini, K. Akarvardar, N. Tripathi, B. Haran, S. Johnson, T. Hook, B. Hamieh, D. Corliss, J. Wang, X. Miao, J. Sporre, J. Fronheiser, N. Loubet, M. Sung, S. Sieg, S. Mochizuki, C. Prindle, S. Seo, A. Greene, J. Shearer, A. Labonte, S. Fan, L. Liebmann, R. Chao, A. Arceo, K. Chung, K. Cheon, P. Adusumilli, H. Amanapu, Z. Bi, J. Cha, H.-C. Chen, R. Conti, R. Galatage, O. Gluschenkov, V. Kamineni, K. Kim, C. Lee, F. Lie, Z. Liu, S. Mehta, E. Miller, H. Niimi, C. Niu, C. Park, D. Park, M. Raymond, B. Sahu, and M. Sankarapandian, “A 7nm FinFET technology featuring EUV patterning and dual strained high mobility channels,” in *2016 IEEE International Electron Devices Meeting (IEDM)*. IEEE, 2016. [Online]. Available: http://dx.doi.org/10.1109/iedm.2016.7838334

[60] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures,” in *Proceedings of the 42nd annual ieee/acm international symposium on microarchitecture*, 2009, pp. 469–480.

[61] S. L. Xi, H. Jacobson, P. Bose, G.-Y. Wei, and D. Brooks, “Quantifying sources of error in mcpat and potential impacts on architectural studies,” in *2015 IEEE 21st International symposium on high performance computer architecture (HPCA)*. IEEE, 2015, pp. 577–589.

[62] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, “Meet the walkers,” *PROC of the 46th MICRO*, pp. 1–12, 2013.

[63] S. Pal, S. Feng, D.-h. Park, S. Kim, A. Amarnath, C.-S. Yang, X. He, J. Beaumont, K. May, Y. Xiong, K. Kaszyk, J. M. Morton, J. Sun, M. O’Boyle, M. Cole, C. Chakrabarti, D. Blaauw, H.-S. Kim, T. Mudge, and R. Dreslinski, “Transmuter: Bridging the efficiency gap using memory and dataflow reconfiguration,” in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 175–190. [Online]. Available: https://doi.org/10.1145/3410463.3414627

[64] S. Kim, M. Fayazi, A. Daftardar, K.-Y. Chen, J. Tan, S. Pal, T. Ajayi, Y. Xiong, T. Mudge, C. Chakrabarti, D. Blaauw, R. Dreslinski, and H.-S. Kim, “Versa: A 36-Core Systolic Multiprocessor With Dynamically Reconfigurable Interconnect and Memory,” *IEEE Journal of Solid-State Circuits*, vol. 57, no. 4, pp. 986–998, 2022.

[65] F. Devaux, “The true processing in memory accelerator,” in *2019 IEEE*

Hot Chips 31 Symposium (HCS). IEEE Computer Society, 2019, pp. 1–24.

[66] S. Diab, A. Nassereldine, M. Alser, J. Gómez-Luna, O. Mutlu, and I. E. Hajj, "A framework for high-throughput sequence alignment using real processing-in-memory systems," *arXiv preprint arXiv:2208.01243*, 2022.

[67] J. Gómez-Luna, I. El Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking a new paradigm: Experimental analysis and characterization of a real processing-in-memory system," *IEEE Access*, vol. 10, pp. 52 565–52 608, 2022.

[68] ARM Limited, "big.little technology: The future of mobile," ARM Holdings, Tech. Rep., 2013, white Paper. [Online]. Available: https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/big-little-technology-the-future-of-mobile.pdf