Unified Thread- and Data-Mapping for Multi-Threaded Multi-Phase Applications on SPM Many-Cores

Vanchinathan Venkataramani
National University of Singapore
vvanchi@comp.nus.edu.sg

Anuj Pathania
National University of Singapore
pathania@comp.nus.edu.sg

Tulika Mitra
National University of Singapore
tulika@comp.nus.edu.sg

***Abstract*—Scratchpad Memories (SPMs) are more scalable than caches as they offer better performance with lower power and area overheads. This scalability advocates their suitability as on-chip memory in many-cores. However, SPM many-cores delegate the responsibility of thread- and data-mapping to the software. The mapping is especially challenging in the case of multi-threaded multi-phase applications. Threads from these applications exhibit both inter- and intra-phase data-sharing patterns. These patterns intricately intertwine thread- and data-mapping across phases. The accompanying qualitative mapping is the key to extract application performance on SPM many-cores.**

State-of-the-art framework for SPM many-cores performs thread- and data-mapping independently. Furthermore, it can only operate with single-phase multi-threaded applications. We are the first to propose in this work, a unified thread- and data-mapping framework for NoC-based SPM many-cores when executing multi-threaded multi-phase applications. Experimental evaluations show, on average, 1.36x performance improvement compared to the state-of-the-art framework for multi-threaded multi-phase applications.

***Index Terms*—SPM, many-core, low-power, task mapping**

I. Introduction

Scratchpad Memories (SPMs), unlike caches, do not provide any hardware-based coherency. SPMs, therefore, have a much lower area and power overheads than traditional caches. These lower overheads make them scalable. This scalability makes them ideal candidates for use as on-chip memories in many-cores with tens or hundreds of cores [1]. However, an SPM also delegates the responsibility for the thread- and data-mapping to the software [2]. An SPM-based many-core has the potential to provide much higher application performance than an equivalent cache-based many-core [3] when provided with an optimized thread- and data-mapping. It also allows for predictable [4] and composable execution [5]. Unsurprisingly, many state-of-the-art many-core architectures such as *Adapteva's Epiphany* [6], *Kalray's MPPA* [7], and *SPECTRUM* [8] now employ SPMs as their on-chip memories.

Figure 1 shows an abstract block diagram for a generalized SPM many-core architecture with a physically distributed on-chip memory. The architecture comprises of several processing tiles connected using a 2D grid Network-on-Chip (NoC) [9]. Each tile contains a processing core, an NoC router, and an SPM. A memory controller(s) provides access to the main memory by interfacing with a tile(s) on the periphery. SPM, unlike caches, do not have a tag array. Therefore, every SPM address space maps to a disjoint portion of the entire memory address space. SPM requires the software to manage the data in its address space. Software is also responsible for moving data from/to off-chip using Direct Memory Access (DMA). Cores can also access data from the remote SPMs in the tiles other than their own via the NoC. Cores execute only one thread at a time and do not support context-switching.

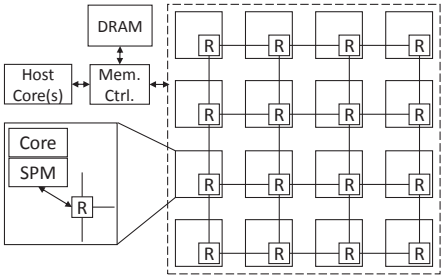
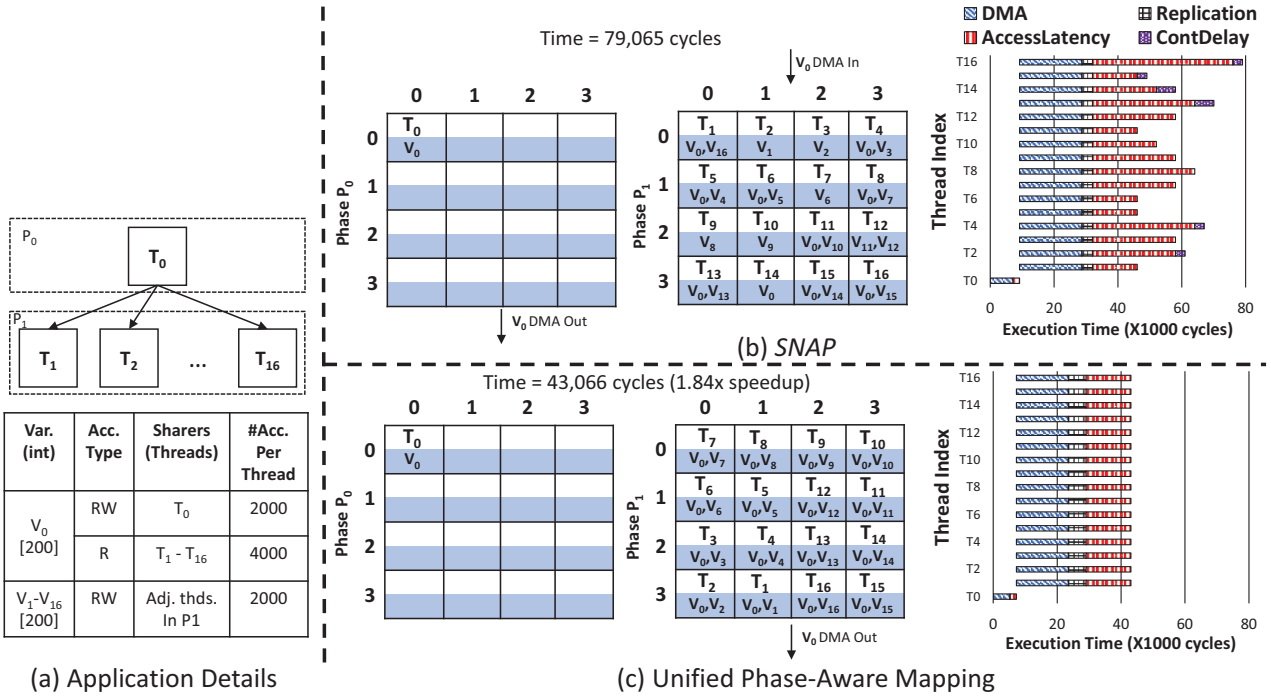


Fig. 1: An abstract block diagram for a generalized NoC-based SPM many-core architecture.

Many-core applications are generally multi-threaded. The threads from these applications exhibit extensive intra-phase data-sharing patterns. These patterns entangle thread- and data-mapping in a complex interdependent relationship. The mapping of threads on cores affects the mapping of data on SPMs and vice-versa. State-of-the-art data-mapping framework [10] for SPM many-cores requires a fixed thread-mapping as an input. Optimization of data-mapping, which is independent of thread-mapping, misses out on significant optimization potential [11]. *In this work, we are the first to present a framework that performs thread- and data-mapping for multi-threaded applications on SPM many-cores in unison.*

Furthermore, several multi-threaded many-core applications now also have multiple phases. Several data variables are common across these interdependent phases. Therefore, the threads from a phase also exhibit extensive inter-phase data-sharing patterns with threads from the subsequent phases. Thread- and data-mapping for a phase, therefore, should also be done per inter-phase data-sharing patterns besides the intra-phase data-sharing patterns. Data-mapping across phases should minimize the DMA overheads involved in moving vari-



Var. (int)	Acc. Type	Sharers (Threads)	#Acc. Per Thread
V_0 [200]	RW	T_0	2000
	R	T_1 - T_{16}	4000
V_1-V_{16} [200]	RW	Adj. thds. In P1	2000

Fig. 2: A motivational example that emphasizes the value of unified thread- and data-mapping for multi-phase multi-threaded applications on SPM many-cores.

ables to/from main memory when switching phases. State-of-the-art data-mapping framework [10] for SPM many-cores can only operate with single-phase multi-threaded applications. Applying this framework in isolation on each phase results in significantly lower performance. *In this work, we are the first to present a framework that performs thread- and data-mapping for multi-phase multi-threaded applications on SPM many-cores.*

Our Novel Contributions. Based on the above discussion, the following are our novel contributions.

- We are the first to present a framework called *UniSPM* that performs unified thread- and data-mapping for multi-phase multi-threaded applications on SPM many-cores.
- We show *UniSPM* provides 1.36x higher performance on average over the state-of-the-art [10] when managing multi-phase multi-threaded many-core applications.

A. Motivational Example

Figure 2 gives an example that emphasizes the importance of unified thread- and data-mapping for multi-threaded multi-phase applications on SPM many-cores. This example uses a 4x4 SPM many-core as hardware with Cores $C_{0,0}$ to $C_{3,3}$ and a multi-threaded multi-phase application A. Figure 1 and Table I gives the system schematics and specifications for the many-core. Figure 2 (a) provides the details of A.

A is a two-phase application with seventeen threads in total. Phase P_0 contains only one thread T_0. Phase P_1 contains sixteen threads T_1 to T_{16}. We assume threads with zero compute for this example. A has seventeen variables in total from V_0 to V_{16}. Thread T_0 produces the variable V_0. Thread T_1 to T_{16} read variable V_0 two thousand times each. Additionally, thread T_x accesses Read-Write variable V_x and $V_{(x+1)\%16}$ four thousand times each, $\forall x \in \{1, 2, \ldots, 16\}$.

Data-mapping framework *SNAP* [10] requires a manually generated thread-mapping for each phase as an input to operate. We provide it with a greedy thread mapping, wherein T_0 (from Phase P_0) maps to core $C_{0,0}$, and thread T_x (from Phase P_1) maps to Core $C_{\lfloor (x-1)/4 \rfloor,(x-1)\%4}$, $\forall x \in \{1, 2, \ldots, 16\}$. Figure 2(b) shows the corresponding data-mapping performed by *SNAP* [10] for application A. *SNAP* performs iterative greedy variable allocation to SPMs, wherein the variable having the highest access density (#accesses/size) in the bottleneck thread is allocated to SPM near to the thread. It decides upon the SPM for allocation based on NoC-latency, NoC-contention, DMA, on-chip copy cost, and variable type. It also performs replication of Read-Only variables to improve performance. A takes 79,065 cycles for execution with *SNAP*.

We can reduce our mapping problem to the Uncapacitated Facility Location Problem (UFLP), which is well-known to be an NP-Hard optimization problem [12]. Therefore, obtaining the optimal task- and data-mapping is computationally expensive. Still, we can obtain the optimal mapping using Integer Linear Programming (ILP) within a reasonable time for our example, given its small design space. We do not show the ILP formulation in this work for brevity. Note that the ILP is not scalable and thereby cannot be used to obtain optimal solutions for real-world applications.

Figure 2(c) shows the optimal unified thread- and data-mapping obtained using an ILP solver. A takes 43,066 cycles for execution with mapping shown in Figure 2(c) with $1.84x$ speedup in comparison to the mapping in Figure 2(b). Re-

duction in variable access latency is responsible for this performance improvement. For example, Figure 2(b) maps the variable V_5 in SPM of core $C_{1,1}$ and threads T_4, T_5 (that access variable V_5) in core $C_{0,3}$ and $C_{1,0}$, respectively. Therefore, latency for threads T_4 and T_5 to access variable V_5 is ten and four cycles, respectively. However, Figure 2(c) maps the threads T_4 and T_5 to adjacent cores $C_{2,1}$ and $C_{1,1}$ respectively, due to their common access of variable V_5 allocated in SPM of Core $C_{1,1}$. Therefore, access latency for variable V_5 in Figure 2(c) for threads T_4 and T_5 is four and one cycle, respectively. A similar access latency decrease for other variables partially explains the performance gain.

Further benefits come from reducing DMA I/O by considering all phases of an application together. Threads from both phases of application A access variable V_0. Figures 2(b) optimizes mapping for each phase in isolation. Therefore, it ends up writing variable V_0 to the main memory at the end of Phase P_0 and then naively bring in a new copy of V_0 using DMA. We can reduce this DMA overhead by leaving variable V_0 on-chip at the end of Phase P_0 for use in Phase P_1. However, both approaches do incur overheads for creating on-chip copies of variable V_0 as the variable access type changes from non-replicable to replicable between phases.

II. Related Work

Several works propose frameworks for efficiently managing program code- and data-mapping for software-controlled SPM in single-cores [13]–[16]. These works target optimal placement using exponential-time algorithms such as 0-1 ILP, graph coloring, etc. However, they are not very scalable and fail to produce results in a reasonable amount of time for SPM many-cores. Recently, authors of [17] propose a genetic algorithm for data-mapping that minimizes DMA overhead for multi-phase single-threaded applications on a single-core.

Authors of [18], [19] propose frameworks for managing SPMs in bus-based multi-cores. Since the SPMs are connected using a bus, these frameworks assume the worst-case latency for accessing any remote SPM. Therefore, these frameworks are oblivious to non-uniform access latency of data sharing in NoC-based SPM many-cores. Authors of [20]–[22] propose frameworks for managing *IBM Cell* micro-architecture based many-cores that also use SPMs. However, all SPMs in *IBM Cell* have per-core private address space, and the hardware itself does not allow any remote SPM access. Therefore, these mechanisms cannot operate on SPM many-cores with shared address space that we explore in this work.

Authors of [10] propose a data-mapping framework for multi-threaded applications on a NoC-based SPM many-core. This framework, however, requires an apriori manual thread-mapping to initiate data-mapping. It is also limited to only single-phase applications. *To the best of our knowledge, ours is the first work that proposes a unified thread- and data-mapping framework for multi-phase multi-threaded applications on a NoC-based SPM many-core.*

III. Problem Formulation

System Model: Let C denote the set of cores (tiles) in the SPM many-core indexed using i. C_i also represents the local SPM of core C_i with capacity $|C_i|$. Let M denote the off-chip DRAM with a capacity large enough to hold all the data. SPM many-core needs to execute a multi-phase multi-threaded application A with P program phases indexed using j.

Thread Model: Every thread within a phase synchronizes on a barrier before proceeding to the next phase. Let T denote the set of application threads across all phases indexed using k. Let $\theta(P_j) \rightarrow T', T' \subseteq T$ denote the set of threads executed in phase P_j. We create dummy buffer threads in a phase if the number of threads in the phase is less than the cores. These buffer threads act as a proxy for storage in SPMs.

Variable Model: Let V denote the set of all variables indexed using l. Let $|V_l|$ denote the size of variable V_l. Let $\psi(V_l, P_j) \rightarrow \{True, False\}$ represent a boolean function that returns true or false when variable V_l in phase P_j is replicable (Read-only) or non-replicable variable (Read-Write or Write-only), respectively. Let $\alpha(T_k, P_j) \rightarrow V'$ and $\beta(T_k, P_j) \rightarrow V'$ denote the set of shared and private variables accessed by thread T_k in phase P_j. Let $\zeta(V_l, T_k)$ denote the number to times thread T_k accesses variable V_l.

Mapping Model: Let $\lambda(T_k, P_j) \rightarrow C_i$ denote the mapping of thread T_k to core C_i in phase P_j. Let $\gamma(T_k, P_j) \rightarrow V'$ denote the set of variables allocated to the core $\lambda(T_k, P_j)$ executing T_k in phase P_j. Let $\gamma(M, P_j) \rightarrow V'$ denote the set of variables allocated to the main memory in phase P_j.

Access Model: Let $\delta(V_l, T_k, P_j)$ denote the total cycles spent by thread T_k across all accesses of variable V_l from the closest memory allocation in phase P_j. $\delta(V_l, T_k, P_j)$ is inclusive of DMA latency for bringing/writing data between off-chip/on-chip, broadcast latency incurred for creating multiple on-chip copies in case of Read-only replicable variables, and NoC latency inclusive of contention delay. Table I states the latency of accessing on-chip SPMs and DMA IO we obtain using micro-benchmarking. We use memory profiling to obtain the number of times thread T_k access variable V_l in phase P_j.

Execution Model: Let $Ecomp(T_k, P_j)$ and $Emem(T_k, P_j)$ represent the computation time and memory access time of thread T_k in phase P_j, respectively.

$$Emem(T_k, P_j) = \sum_{\forall_{V_l \in \alpha(T_k,P_j) \cup \beta(T_k,P_j)}} \delta(V_l, T_k, P_j) \quad (1)$$

Let $E(T_k, P_j)$ represent the execution time of thread T_k in phase P_j. $E(T_k, P_j)$ is the sum of $Ecomp(T_k, P_j)$ and $Emem(T_k, P_j)$.

$$E(T_k, P_j) = Ecomp(T_k, P_j) + Emem(T_k, P_j) \quad (2)$$

Let $E(P_j)$ represent the execution time of phase P_j. Since all threads in a phase synchronize on a barrier, execution time of the bottleneck thread determines $E(P_j)$.

$$E(P_j) = \max_{\forall_{T_k \in \theta(P_j)}} E(T_k, P_j) \quad (3)$$

Let E represent the execution time of application A. E is the sum of the execution time of all phases.

$$E = \sum_{\forall_{P_j \in P}} E(P_j) \tag{4}$$

Computation time $Ecomp(T_k, P_j)$ is fixed and independent of mapping. Memory access time $Emem(T_k, P_j)$ is dependent upon both thread- and data-mapping. Therefore, we focus on minimizing memory access time by optimizing the mapping.

Constraints: SPM many-core enforces three constraints on the thread- and data-mapping. (1) Capacity Constraint: Size of the total variables allocated to an SPM should be less than than the size of SPM.

$$\left(\sum_{\forall_{V_l \in \gamma(T_k, P_j)}} |V_l| \right) \leq |\lambda(T_k, P_j)| \tag{5}$$

(2) Uniqueness Constraint: There is no coherence in SPM many-cores. Therefore, a non-replicable variable in a phase must exist in only one place in memory during the phase.

$$V_l \in \Big(\big(\cap_{\forall_{T_k \in \theta(P_j)}} \gamma(T_k, P_j) \big) \cap \gamma(M, P_j) \Big) \iff \psi(V_l, P_j) = True \tag{6}$$

(3) One-Thread-Per-Core Constraint: There is a one-to-one mapping between threads and cores within a phase. Every thread active in a phase must be assigned a core.

$$\forall_{T_k, T_{k'} \in \theta(P_j)} \ \lambda(T_k, P_j) \neq \lambda(T_{k'}, P_j) \neq \emptyset \tag{7}$$

Objective: The objective is to minimize application execution time E by determining the output of thread- and data-mapping function λ and γ, respectively. Mapping functions must respect the constraints given in Equations 5, 6, and 7.

IV. Proposed UniSPM Framework

We propose a logarithmically recursive framework called *UniSPM* to heuristically solve the NP-hard unified thread- and data-mapping problem for multi-threaded multi-phase applications on SPM many-cores. Algorithm 1 gives the pseudo-code for the algorithm we use in *UniSPM*. The goal of *UniSPM* is to minimize the flow of on-chip data as well as off-chip data. Since the cost of memory I/O is significantly more than remote SPM I/O (Table I), *UniSPM* prioritizes the sequestration of off-chip data flow over on-chip data flow.

Equation 7 dictates that *UniSPM* must allocate a core to every thread in a phase. *UniSPM* determines the core wherein a thread is mapped to in a phase at the end of recursion only. In intermediate recursions, *UniSPM* virtually assigns sets of variables to set of threads without actually specifying which thread in the set of threads will finally hold the variable in the end. Let $\bar{\gamma}(T', P_j) \rightarrow V'$ represent a set of variables $V' \subseteq V$ *UniSPM* hypothetically assigns to a set of threads $T' \subseteq T$ in phase P_j. Note, $\gamma(T_k, P_j)$ that defines real variable assignment to a thread stands in orthogonally to $\bar{\gamma}(T', P_j)$. The size of variables assigned to a set of threads in a phase (both in reality and hypothetically) can act as a good proxy to determine the remaining capacity of the SPMs of the set of cores where *UniSPM* eventually maps the threads. Let $\phi(T', P_j)$ represent the free space remaining with a set of threads $T' \subseteq T$ in phase P_j for new variable assignment.

Algorithm 1 Algorithm in *UniSPM*

```
1: function INITIATEMAP(C, T, V)
2:     ∀_{P_j∈P} ∀_{T_k∈θ(P_j)} γ(T_k, P_j) = γ(T_k, P_j) ∪ β(T_k, P_j)
3:     ∀_{P_j∈P} ∀_{T_k∈θ(P_j)} γ̄(T_k, P_j) = ∅
4:     ∀_{P_j∈P} RECURSIVEMAP(C, θ(P_j), V, P_j)
5:     return γ(T_k, P_j), λ(T_k, P_j)
6: end function
7: function RECURSIVEMAP(C,T,V,P_j)
8:     if |C| = 1 then
9:         λ(T_k ∈ T, P_j) = C_i ∈ C
10:        FIT (γ(T_k, P_j), γ̄(T_k, P_j))          ▷ Hypothetic to Real Assignment
11:        return                                    ▷ Break Recursion
12:    end if
13:    BIFURCATE (C) → {C_L, C_R}      ▷ Equal Rectangular or Square Division
14:    BIFURCATE (T) → {T_L, T_R}      ▷ Using K-Mean Clustering Variation
15:    V = ↓ ∪_{∀_{T_k∈T}} α(T_k, P_j)          ▷ Decreasing Access Density Sort
16:    for V_l ∈ V do
17:        if ζ(V_l, T_L) ≥ ζ(V_l, T_R) & φ(T_L, P_j) ≥ |V_l| & V_l ∉ γ̄(T_L, P_j) then
18:            γ̄(T_L, P_j) = γ̄(T_L, P_j) ∪ V_l
19:        else if φ(T_R, P_j) ≥ |V_l| & V_l ∉ γ̄(T_R, P_j) then
20:            γ̄(T_R, P_j) = γ̄(T_R, P_j) ∪ V_l
21:        end if
22:    end for
23:    Synchronize ∀_{P_j∈P}   ▷ Wait for Allocation for All Phases Before Replication
24:    REPLICATION (T_L, T_R) → {T_L, T_R}
25:    RECURSIVEMAP (C_L,T_L,V,P_j)
26:    RECURSIVEMAP (C_R,T_R,V,P_j)
27: end function
```

UniSPM performs the following procedure for all phases in parallel in Algorithm 1. It begins by first assigning (in real) thread's private variables in its local SPM (Line 2). *UniSPM* then bifurcates the cores (Line 13) to be allocated in a phase into two equal rectangular sets of cores. It also divides the threads (Line 14) to be allocated in that phase into two equal sets of threads using same-size k-means clustering. A thread clustering sub-routine places the thread onto a multi-dimensional space. Sub-routine assigns one dimension in the multi-dimensional space to each unique shared variable accessed by the threads in the phase. Sub-routine assigns vector coordinates to each thread in the phase based on its access density for shared variables. The use of access density normalizes the distance in different dimensions. Distance in the space is then used to bifurcate threads in a phase into two equal-size sets of threads. In the end, *UniSPM* has two sets of threads for each phase. Algorithm 1 does not show the pseudo-code for clustering for brevity.

UniSPM then sorts the shared variables based on access density and then assesses them for on-chip allocation in descending order (Line 15). It then assigns shared variables to on-chip SPMs by assigning them to either of the two sets of threads depending upon which cluster makes more aggregate accesses (Lines 16-22). A set of threads must have enough space to accommodate the variable in its remaining free space. The variable must also be unique to the set. If one set of threads gets full, *UniSPM* assigns the shared variable to the other set of threads. If a variable is too big to fit in either of the clusters, then *UniSPM* skips it. *UniSPM* considers all shared variables for on-chip allocation as long as they can fit in the remaining space. *UniSPM* then does one-to-one association of two sets of threads to two sets of cores for each phase using two distinct recursive calls (Lines 25 and 26).

Once *UniSPM* finishes the above variable allocation for all sets of threads in all phases (Line 23), it attempts to further bring down the memory access time with the help of inter- and intra-phase replication using any remaining free space in either set of the threads (Line 24). The goal of the inter-phase replication is to avoid DMA to main memory by retaining a variable from an earlier phase on-chip in intermediate phases if a later phase uses it. The location of this inter-phase variable is inconsequential, but all the intermediate phases must retain it on-chip to successfully avoid the DMA call. *UniSPM* creates a list of all variables, which are brought into on-chip memory using DMA more than once. It then sorts them according to their DMA cost (#DMACalls). It then assigns the variable to one set of threads from each phase in all the intermediate phases, where it is missing, such that at least one DMA I/O is avoided. It assigns the variable to the set of threads with most remaining space if the assignment is possible on either set of threads. The variable is not replicated at all if there is not enough space in any of the intermediate phases. Intra-phase replication only happens if there is still space remaining after inter-phase replication on either set of threads. The goal of intra-phase replication is to minimize the remote SPM access latency by on-chip replication of replicable shared variables. *UniSPM* only considers the variables it did not previously assign to a set of threads for replication in the set of threads. It then starts assigning variables to the sets of threads in descending order of aggregate accesses if there is enough free space available in them. Algorithm 1 does not show the pseudo-code of replication for brevity.

UniSPM recursively bifurcates set of cores and threads logarithmically, and continue to perform one-to-one associations between them. The last recursion provides us with the final thread- and data-mapping for each core in each phase (Lines 8-11). It may happen that due to imperfect bin-packing with discrete-size variables and SPMs, *UniSPM* may have to thrash some hypothetically assigned variables during the final real data-mapping on to the core (Line 10). In such a case, *UniSPM* thrashes inter-phase replicated variables first, followed by intra-phase replicated variables (across all intermediate phases simultaneously), and then the shared variables. Within the variables of the same type, *UniSPM* thrashes them in increasing access density order.

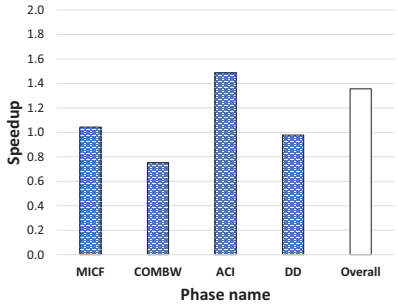
Computational Complexity: *UniSPM* recursively searches $O(|C|)$ nodes, where $|C|$ is the number of cores in the many-core. *UniSPM* explores all P phases in parallel (Line 4) in each search adding in a factor of $O(|P|)$ to the complexity, where $|P|$ is the number of phases. $|V|$ and $|T|$ denotes the number of variables and threads, respectively. *UniSPM* performs K-means clustering (Line 14) and access density sort (Line 15) in each search with complexity $O(|V||T| + |T| \ln |T|)$ and $O(|V| \ln |V|)$, respectively. All the other operations that *UniSPM* performs serially in the search have lower complexity than K-means clustering and access density sort. Therefore, the worst-case computation complexity of *UniSPM* is $O(|P||V|(|V||T| + |T| \ln |T| + |V| \ln |V|))$.

TABLE I: System specifications for *Parallella* embedded development platform with *Epiphany* SPM many-core.

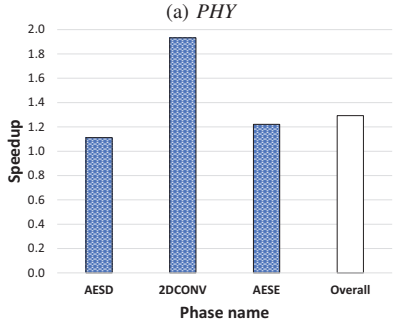
Cores	2 *ARMv7* host cores 16 Epiphany in-order (dual-issue) cores, 600 MHz
SPM	Unified I & D, 32 KB, 4 banks, 1-cycle access latency
Network	2D Mesh, 1.5 cycle per hop latency, XY routing
Memory	1 GB, 500-cycle access latency
DMA data transfer rate	on-chip: write 1236.81 MB/s, read 392 MB/s off-chip: write 234.35 MB/s, read 87.71 MB/s

TABLE II: Problems solving time with different benchmarks.

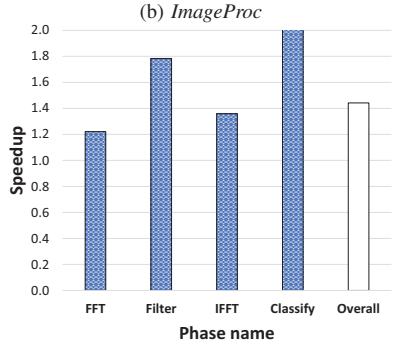
	#Ph.	#Thd.	#Var.	Data Size (in KB)	Solving Time (in sec.)
PHY	4	44	134	599	0.29
ImageProc	3	48	70	66	0.21
Gesture	4	24	41	120	0.27



(a) *PHY*



(b) *ImageProc*



(c) *Gesture*

Fig. 3: Speedup attained with *UniSPM* against *SNAP* for different benchmarks and their individual phases.

V. Experimental Evaluation

Setup: Table I summarizes the specification of the *Epiphany* platform with an SPM many-core. *Epiphany* contains a unified SPM for both instructions and data. We utilize 16 KB for instructions and stack variables. We allocate the remaining 16 KB to data variables. *Epiphany* lacks compiler and library

support for multi-phase many-core applications for direct execution. Therefore, we use a memory simulator based on *Epiphany* platform to evaluate the different frameworks.

Baseline: We compare *UniSPM* proposed in this work with *SNAP* [10]. *SNAP* iteratively tries to reduce the execution time of the critical thread by allocating the variable with the highest access density to a closer location, such that application execution time does not increase. *SNAP* creates multiple copies if the reduction in execution time is more than the overhead incurred in making an additional on-chip copy. Since *SNAP* relies on manual task mapping, we utilize a greedy task mapping for our evaluation. Additionally, it writes/reads data between off-chip and on-chip after/before every phase. In *UniSPM*, we propose a unified task- and data-mapping that considers both inter- and intra-phase data sharing.

Benchmarks: We use three multi-threaded multi-phase applications – *PHY* [23], *ImageProc* [24], and *Gesture* [25] as benchmarks. *PHY* is a Long Term Evolution (LTE) baseband processing application containing four phases – *MICF*, *ACI*, *COMBW*, and *DD*. *ImageProc* is an Image Processing Internet of Things (IoT) application consisting of three phases – *AESD, 2DCONV*, and *AESE*. *Gesture* is a Human-Computer Interface (HCI) application consisting of four phases – *FFT, Windowing, IFFT*, and Classification. We utilize worst-case memory profiling and static analysis (to find the variable access type) as input to obtain the task- and data-mapping using *UniSPM*. Therefore, the performance may vary with the size of the input, but the functionality is never wrong.

Performance: Figure 3 shows the speedup (reduction in memory access time) for different benchmarks (and their phases) with *UniSPM* against *SNAP*. As seen in this figure, *UniSPM* results in 1.36x, 1.29x, and 1.44x speedup concerning *SNAP* for *PHY*, *ImageProc*, and *Gesture*, respectively, and an overall average of 1.36x. Note that in *COMBW* phase of *PHY*, *UniSPM* thrashes variables to off-chip due to imperfect bin-packing. However, *UniSPM* still improves the overall time as other phases dominate the time.

Scalability: *UniSPM* uses a polynomial-time algorithm to perform mapping and thereby scales well with an increase in the number of phases, threads, and variables. Table II gives the problem-solving time for different benchmarks with *UniSPM*. *UniSPM*, on average, takes only 0.26 seconds to map.

VI. Conclusion

In this work, we proposed the first-ever framework (called *UniSPM*) to perform unified thread- and data-mapping for multi-threaded multi-phase applications executing on SPM many-cores. *UniSPM* uses a low-overhead recursive algorithm to solve the mapping problem. Empirical evaluations reveal 1.36x performance improvement compared to the state-of-the-art mechanism for multi-threaded multi-phase applications.

VII. Acknowledgment

This work was supported by the National Research Foundation, Prime Ministers Office, Singapore under its Industry-IHL Partnership Grant NRF2015-IIP003.

References

[1] J. Teich *et al.*, "Invasive Computing: An Overview," *Multiprocessor System-on-Chip*, 2011.
[2] S. Wasly *et al.*, "A Dynamic Scratchpad Memory Unit for Predictable Real-Time Embedded Systems," in *Euromicro Conference on Real-Time System (ECRTS)*, 2013.
[3] M. Rapp *et al.*, "Prediction-Based Task Migration on S-NUCA Many-Cores," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019.
[4] D. Gangadharan *et al.*, "Application-Driven Reconfiguration of Shared Resources for Timing Predictability of MPSoC Platforms," in *Asilomar Conference on Signals, Systems and Computers (ACSCC)*, 2014.
[5] A. Molnos *et al.*, "A Composable, Energy-Managed, Real-Time MPSOC Platform," in *Optimization of Electrical and Electronic Equipment (OPTIM)*, 2010.
[6] A. Olofsson, "Epiphany-V: A 1024 Processor 64-bit Risc System-on-Chip," *arXiv preprint arXiv:1610.01832*, 2016.
[7] B. D. de Dinechin, "Kalray MPPA®: Massively Parallel Processor Array: Revisiting DSP Acceleration with the Kalray MPPA Manycore Processor," in *Hot Chips Symposium (HCS)*, 2015.
[8] V. Venkataramani *et al.*, "SPECTRUM: A Software Defined Predictable Many-Core Architecture for LTE Baseband Processing," *Languages Compilers, Tools and Theory of Embedded Systems (LCTES)*, 2019.
[9] S. M. PD *et al.*, "A Scalable Network-on-chip Microprocessor with 2.5 D Integrated Memory and Accelerator," *Transactions on Circuits and Systems I: Regular Papers (TCAS I)*, 2017.
[10] V. Venkataramani *et al.*, "Scratchpad-Memory Management for Multi-Threaded Applications on Many-Core Architectures," *Transactions on Embedded Computing Systems (TECS)*, 2019.
[11] M. Xu *et al.*, "Holistic Resource Allocation for Multicore Real-Time Systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.
[12] K. Jakob *et al.*, "The Simple Plant Location Problem: Survey and Synthesis," *European Journal of Operational Research (EJOR)*, 1983.
[13] N. Nguyen *et al.*, "Memory Allocation for Embedded Systems with a Compile-Time-Unknown Scratch-Pad Size," *Transactions on Embedded Computing Systems (TECS)*, 2009.
[14] S. Udayakumaran *et al.*, "Dynamic Allocation for Scratch-pad Memory Using Compile-time Decisions," *Transactions on Embedded Computing Systems (TECS)*, 2006.
[15] A. Dominguez *et al.*, "Heap Data Allocation to Scratch-pad Memory in Embedded Systems," *Journal of Embedded Computing.*, 2005.
[16] F. Angiolini *et al.*, "A Post-Compiler Approach to Scratchpad Mapping of Code," in *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2004.
[17] M. R. Soliman *et al.*, "WCET-Driven Dynamic Data Scratchpad Management With Compiler-Directed Prefetching," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2017.
[18] M. Verma *et al.*, "Scratchpad Sharing Strategies for Multiprocess Embedded Systems: A First Approach," in *Embedded Systems for Real-Time Multimedia (ESTImedia)*, 2005.
[19] V. Suhendra *et al.*, "Integrated Scratchpad Memory Optimization and Task Scheduling for MPSoC Architectures," in *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2006.
[20] J. Lu *et al.*, "SSDM: Smart Stack Data Management for Software Managed Multicores (SMMs)," in *Design Automation Conference (DAC)*, 2013.
[21] J. Lu et al., "Efficient Code Assignment Techniques for Local Memory on Software Managed Multicores," *Transactions on Embedded Computing Systems (TECS)*, 2015.
[22] K. Bai *et al.*, "Automatic and Efficient Heap Data Management for Limited Local Memory Multicore Architectures," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2013.
[23] M. Sjalander *et al.*, "An lte uplink receiver phy benchmark and subframe-based power management," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2012.
[24] C. Tan *et al.*, "Locus: Low-power customizable many-core architecture for wearables," in *IEEE International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2016.
[25] C. Tan *et al.*, "Stitch: Fusible heterogeneous accelerators enmeshed with many-core architecture for wearables," in *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.