Mix and Match: Reorganizing Tasks for Enhancing Data Locality

XULONG TANG, University of Pittsburgh, USA
MAHMUT TAYLAN KANDEMIR, Penn State University, USA
MUSTAFA KARAKOY, TUBITAK-BILGEM, Turkey

Application programs that exhibit strong locality of reference lead to minimized cache misses and better performance in different architectures. However, to maximize the performance of multithreaded applications running on emerging manycore systems, data movement in on-chip network should also be minimized. Unfortunately, the way many multithreaded programs are written does *not* lend itself well to minimal data movement.

Motivated by this observation, in this paper, we target task-based programs (which cover a large set of available multithreaded programs), and propose a novel compiler-based approach that consists of four complementary steps. First, we partition the original tasks in the target application into sub-tasks and build a data reuse graph at a sub-task granularity. Second, based on the intensity of temporal and spatial data reuses among sub-tasks, we generate new tasks where each such (new) task includes a set of sub-tasks that exhibit high data reuse among them. Third, we assign the newly-generated tasks to cores in an architecture-aware fashion with the knowledge of data location. Finally, we re-schedule the execution order of sub-tasks within new tasks such that sub-tasks that belong to different tasks but share data among them are executed in close proximity in time.

The detailed experiments show that, when targeting a state of the art manycore system, our proposed compiler-based approach improves the performance of 10 multithreaded programs by 23.4% on average, and it also outperforms two state-of-the-art data access optimizations for all the benchmarks tested. Our results also show that the proposed approach i) improves the performance of multiprogrammed workloads, and ii) generates results that are close to maximum savings that could be achieved with perfect profiling information. Overall, our experimental results emphasize the importance of dividing an original set of tasks of an application into sub-tasks and constructing new tasks from the resulting sub-tasks in a data movement- and locality-aware fashion.

CCS Concepts: • **Computer systems organization → Multicore architectures**; • **Software and its engineering → Compilers**.

Additional Key Words and Phrases: parallelism; code transformation; compiler optimization

ACM Reference Format:
Xulong Tang, Mahmut Taylan Kandemir, and Mustafa Karakoy. 2021. Mix and Match: Reorganizing Tasks for Enhancing Data Locality. *Proc. ACM Meas. Anal. Comput. Syst.* 5, 2, Article 20 (June 2021), 24 pages. https://doi.org/10.1145/3460087

Authors' addresses: Xulong Tang, tax6@pitt.edu, University of Pittsburgh, Pittsburgh, PA, USA; Mahmut Taylan Kandemir, mtk2@psu.edu, Penn State University, State College, PA, USA; Mustafa Karakoy, m.karakoy@yahoo.co.uk, TUBITAK-BILGEM, Turkey.

2476-1249/2021/6-ART20 $15.00
https://doi.org/10.1145/3460087

1 INTRODUCTION

Data access performance plays an important role in shaping the overall performance of applications running on multicore and manycore systems. One of the important factors that affect data access performance is the “amount of data movement” involved in serving the data from the location where they reside to the node where they are requested. While modern manycores generally employ scalable on-chip networks (e.g., two-dimensional mesh) to reduce design complexity, such type of network topology increases the data accesses latencies as well as the latency variability. In particular, if there is a long distance between the requesting node and data location, the access will experience more time due to i) more on-chip network links to traverse and ii) increased network contention.

There exists a substantial body of prior research efforts including data locality-oriented cache optimizations [6, 11, 22, 33, 53, 57, 61], data layout transformations [23, 35, 37, 39, 48, 59], and near-data computing (NDC) techniques [7, 9, 19, 21, 24, 29, 50], aiming to reduce the cost of data accesses in single-core and manycore systems. Among them, NDC is one popular execution paradigm that offloads computations to execute near data, instead of the traditional approaches that fetch data to computation. While NDC effectively reduces data movements and improve application performance, most NDC approaches require *hardware modifications* such as extra compute units [21, 25, 42] and those near memory computing units usually have limited computing capabilities and can only process simple operations (e.g., addition and transpose). Compared to NDC, a more recent approach, called computing with near data (CND) [2, 5, 49], re-generates the requested costlier (remote) data using less-costlier (nearby) data to avoid high data movement cost. However, CND is not free either; in particular, it requires heavy program analysis and annotations to guarantee correctness and effectiveness, which makes it difficult to employ for complex application programs.

In this paper, we propose another strategy for optimizing data accesses without the need of any hardware modification (consequently, it can be used in any existing manycore architecture). Our approach, focusing on applications whose parallelism can be expressed using a “static task graph”, is built upon the intuition that, computation among tasks with inherent data sharing should be clustered and scheduled to execute considering data reuse opportunities from both “spatial” dimension and “temporal” dimension. To be more concrete, for the “spatial” dimension, computations with inherent data sharing should be executed on nearby cores in an on-chip network based manycore system such that the requested data can be supplied from nearby locations without going to far away locations. On the other hand, from the perspective of “temporal” dimension, computations with data sharing among them should execute in close proximity in time so that the requested data can be found in nearby cores’ caches. Since original tasks are typically heavy and consist of many operations and data accesses, it is difficult to effectively capture and exploit data reuse opportunities from both the “spatial” dimension and “temporal” dimension at a task granularity. To this end, we first partition the tasks in a given parallel application into finer-granular units, called *sub-tasks*, and cluster sub-tasks into *new* tasks such that the sub-tasks with inherent data reuse among them i) are clustered into new tasks which are assigned to nearby cores, and ii) are executed in an order to capture temporal data reuse. Meanwhile, load-balance across cores are maintained *without* compromising the degree of parallelism. In fact, one additional benefit from our approach is that the degree of parallelism in the application can increase, in most cases, compared to original task-level parallelism. The main contributions of this paper are as follows:

- We observe that there exist two major performance deficiencies in current task-centric parallelism. First, tasks consist of heavy computation that generates a large memory footprint. Second, current task scheduling mechanisms are agnostic to the underlying architecture organization, and fail to leverage the data reuse opportunities originating from neighboring nodes. Consequently, it leads to massive on-chip network data movement that degrades the data access performance and also diminishes the degree of parallelism.

- We propose a novel compiler-based approach with the goal of reducing the on-chip data movements for task-parallel applications on manycore systems. Our approach consists of four complementary steps. First, we partition the original tasks of the target application into *sub-tasks*, and build a data reuse graph at a sub-task granularity. Second, based on the intensity of the inherent data reuse, our approach forms *new tasks* where each such (new) task includes a set of sub-tasks (coming from the original tasks) that exhibit data reuse among them. Third, we assign the newly-formed tasks to cores in an *architecture-aware fashion* with the knowledge of data location. Finally, we re-schedule the execution order of the sub-tasks within new tasks such that the sub-tasks that share data among them execute closer in time. As a result, since each newly-formed task exhibits good locality, on-chip cache performance is improved, and both frequency and volume of the message traffic on the on-chip network are reduced.
- We conduct a detailed experimental evaluation using a state-of-the-art manycore platform and three different benchmark suites. Our experimental evaluations with one of these benchmark suites reveal that the proposed optimization strategy i) improves the performance of all 10 multithreaded programs tested (23.4% on average), ii) outperforms two previously-proposed state-of-the-art data accesses optimization strategies (one compiler and one hardware based) in all benchmarks tested, iii) is effective with multiprogrammed workloads as well, and iv) generates results that are close to maximum savings that could be achieved with perfect profile information. Furthermore, our evaluations with the other two benchmark suites indicate 23.3% and 19.3% average performance improvements brought by our approach, due to improved data locality and reduced data movement. Our results emphasize that, for minimum amount of data movement on the on-chip network and maximum data locality enhancement, *both* sub-task relocation and sub-task rescheduling need to be performed.

The rest of this paper is organized as follows. The next section describes the manycore architecture targeted by this work, and Section 3 explains the baseline task model. A high-level view of our approach is presented in Section 4, and its technical details are given in Section 5. Section 6 gives an experimental evaluation of the proposed optimization strategy, and Section 7discusses related work. Finally, the paper is concluded in Section 8 with a summary of our major contributions.

2 TARGET ARCHITECTURE

In this paper, we target on-chip network (NoC) connected manycore systems. Figure 1 shows the high-level NoC topology of one such system – Intel's Knights Landing (KNL) [45]. While our approach discussed in this paper focuses on 2D mesh-based architectures, it is applicable to any type of NoC, as long as it is exposed to the compiler. KNL has 36 tiles connected through a mesh network. Inside each tile, there are two cores and each core features two 512-bit AVX vector units (VUs) as well as a per-core private L1 cache. A directory-based cache coherence is maintained among L2 caches across tiles. KNL is also equipped with a 16 GB multi-channel dynamic random access memory (MCDRAM) split into 8 channels and connected to the four corners of the mesh. At boot time, this MCDRAM can be configured in one of three modes: cache mode, flat mode, and hybrid mode. Specifically, the MCDRAM is configured as a conventional last-level cache (LLC) in cache mode, whereas in flat mode the MCDRAM is configured as an addressable memory extension. In hybrid mode, one can configure 25% (50%) of the MCDRAM capacity to LLC and the remaining capacity to addressable memory. KNL also employs three different cluster modes which divide the mesh into different virtual regions to keep the on-chip network data movement as local as possible.

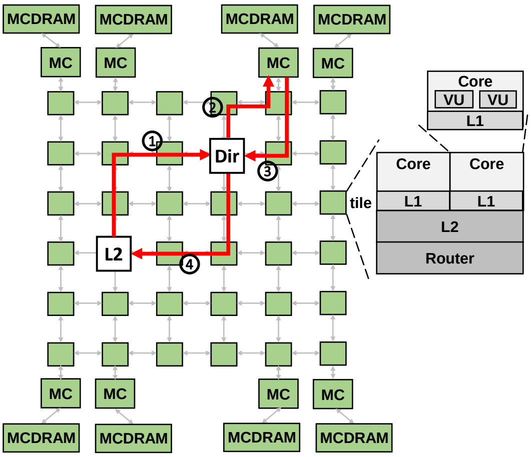


Fig. 1. High level architecture of Intel Knight's Landing (KNL) processor.

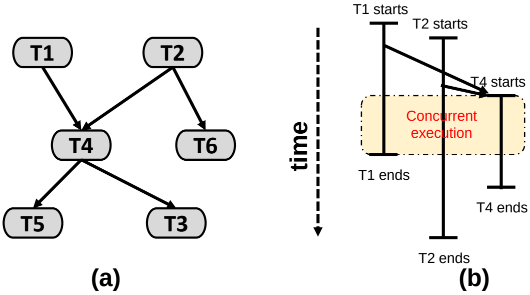
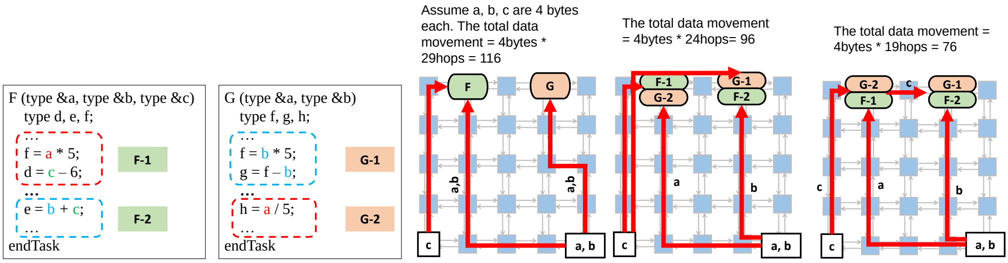


Fig. 2. (a) An example of static task graph where dependencies are labeled with arrows. (b) Concurrent execution across tasks when data dependencies are resolved.

In the "all-to-all" mode, the addresses are uniformly spread over the caches and memory controllers. In the "quadrant" mode, memory accesses only travel within the same virtual region, and finally, in the "SNC-4" mode, the mesh is split into 4 non-uniform memory access (NUMA) clusters.

Figure 1 also depicts a typical "data access path" with L2 cache coherence. Specifically, if a data access misses in L2 cache of local tile, the request is forwarded to the corresponding cache directory node (①). If it hits in another tile's L2 cache, the requested data is forwarded from that tile and the corresponding entry in the directory is updated. Otherwise, if the request misses in L2 caches, a memory access request is generated and forwarded to the corresponding memory controller (②). Finally, the requested data is fetched from the memory and sent to the requested tile. The data movement involved in this data fetch comes from two scenarios. First, if the requested data reside in another tile's L2 cache, it will be forwarded to the requesting tile. Given the mesh connection, if there are many network links (hops) between these two tiles, the corresponding data movement will span a long distance. Second, if the request misses from L2 caches, the requested data will be supplied by corresponding memory controller. Clearly, the shorter distance between the memory controller and the requesting tile, the less data movement. In this paper, we target both the scenarios and our proposed approach reduces data movements in both the cases.



```
F (type &a, type &b, type &c)
   type d, e, f;
   …
   f = a * 5;
   d = c – 6;
   …
   e = b + c;
   …
endTask
```

```
G (type &a, type &b)
   type f, g, h;
   …
   f = b * 5;
   g = f – b;
   …
   h = a / 5;
   …
endTask
```

Assume a, b, c are 4 bytes each. The total data movement = 4bytes * 29hops = 116

The total data movement = 4bytes * 24hops= 96

The total data movement = 4bytes * 19hops = 76

(a) Code example (b) Default scenario (c) Clustering (d) Scheduling

Fig. 3. An example illustrating how our proposed approach reduces data movements.

3 TASK GRAPH

In this paper, we focus on "task-centric" parallelism in the broadest sense of the term. In this context, the focus is on distributing tasks, which can be concurrently performed by threads, across available cores. A task can be a function, a series of functions, a loop nest, or a subset of iterations of a loop nest. For example, in an OpenMP [40] application, each iteration of a parallel loop can be considered as a task; or, if desired, a set of neighboring iterations of a parallel loop can be considered as a task. As such, our task concept is quite general and in fact many existing multi-threaded programs written in different languages/libraries can be cast as task-centric (not only pThreads or OpenMP based ones but also, for example, TBB [56] and CilkPlus [43] based ones as well).

Figure 2(a) shows an example with six tasks. Each node in the graph represents a task and the edges capture the data and control dependencies among the tasks.[1] "Synchronization primitives" are used to guarantee that a task starts only when certain conditions are satisfied. For instance, in Figure 2, tasks $T1$ and $T2$ can execute in parallel, whereas $T4$ needs to wait for both $T1$ and $T2$, due to data or control dependencies. It is important to emphasize however that, in this task graph-based execution, portions from two tasks can execute concurrently in some cases even if there exist data dependencies among them. For example, Figure 2(b) shows one possible execution timeline of tasks $T1$, $T2$, and $T4$. As shown in the figure, there exists data dependency between $\{T1, T2\}$ and $T4$ in Figure 2(a). However, $T4$ can start its execution without waiting $T1$ or $T2$ to finish as long as its input data have already been generated by $T1$ and $T2$. In other words, $T4$ can execute concurrently with $T1$ and $T2$ after the data dependency has been resolved. Similar situation exists between $T2$ and $T4$. As a result, all three tasks have overlapped execution in timeline as shown in Figure 2(b). Note that, such execution paradigm is adopted in various modern task-based parallelization models as well, e.g., "depend clause" in OpenMP task primitives [40].

4 HIGH LEVEL VIEW OF OUR APPROACH

Targeting the static task graph-based executions, we ask a fundamental question when these tasks are executed by different threads on multiple different cores: *can we execute them in a "data reuse-aware" fashion?* To this end, our goal in this paper is to organize computation in task-centric programs such that i) the amount of data movement is reduced, and ii) the data locality is improved.

Let us consider the example shown in Figure 3. Figure 3a gives the code of two functions, $F()$ and $G()$. Let us consider F and G as two independent "tasks" that are to be executed by two different cores, and they share data structures a and b. In the default execution scenario, shown in Figure 3b,

[1] Note that, conceptually, any two code portions within two different tasks can execute in parallel if they do not depend on each other.

even if cache coherence is provided (discussed in Section 2), since F fetches a at the beginning of its execution and b towards the end of its execution, and G fetches b at the beginning and a towards the end, it is quite likely that a would be *evicted* from F's cache by the time G tries to fetch it. A similar situation happens for the shared data structure b as well. As a result, *both* these tasks, F and G, would fetch a and b from off-chip memory, resulting in a total amount of 116 data movements for fetching a, b, and c (the calculation is given in Figure 3b).

A major problem in this default execution is that, it fails to leverage the "spatial" data reuse opportunities. To be more specific, the inherent data reuse in this example is *separated* into two different tasks which are scheduled to execute on two different cores. To exploit the spatial data reuse opportunities, our approach splits tasks into finer granular units, referred to as *sub-tasks* in this paper, and clusters the resultant sub-tasks into *new* tasks such that the sub-tasks with intensive data reuse among them are co-located and scheduled in the same core for execution. Figure 3c shows the execution after our clustering. In particular, $F - 1$ and $G - 2$ sharing a are co-located in the same core, and similarly, $F - 2$ and $G - 1$ sharing b are co-located together. As a result, the total number of data movements needed for fetching a, b, and c is reduced to 96.

One potential problem brought by sub-task clustering in this particular example is that it introduces extra data movements for c. In the default/baseline execution, c is only fetched by F. However, with our sub-task clustering, c is fetched twice by $F - 1$ and $F - 2$ which execute on two different cores. Even with cache coherence in place, c is unlikely to be found in the cache of the node where $F - 1$ executes. This is because $F - 2$ executes after $G - 1$ finishes, and by the time $F - 2$ fetches c, it may have already been evicted from the cache of the node where $F - 1$ executes. In other words, although sub-task clustering reduces the total data movement and benefits a and b, it does *not* capture the "temporal" data reuse opportunity of c. To address this problem, we *re-schedule* the sub-tasks within a newly-generated task. Figure 3d illustrates the result after re-scheduling. Specifically, we switch the execution orders of $F - 1$ and $G - 2$ such that $F - 1$ and $F - 2$, which share c, will execute in close proximity in time. As a result, the resulting code has a higher probability that c would be forwarded from one node's cache based on the underlying coherence protocol. After this re-scheduling, the total data movement is further reduced to 76. This example demonstrates that, *for minimum amount of data movement on the on-chip network and maximum data locality, both sub-task relocation and sub-task rescheduling have to be performed.*

More generally, our proposed approach *forms* new tasks from original tasks, by dividing the original tasks into sub-tasks and putting the sub-tasks that exhibit intensive data reuse among them into a same new task. Since each newly-formed task will eventually be mapped to a single node for execution, the reuse among its sub-tasks will be captured by the local caches in that node, that is, the *data reuse* among those sub-tasks will be converted into *data locality.* Figure 4(a) shows an example with four original tasks ($Task - 1$, $Task - 2$, $Task - 3$, and $Task - 4$). Each original task consists of several sub-tasks (denoted using cycles). For example, $Task - 1$ has five sub-tasks labeled from $T11$ to $T15$. The data sharing (data reuse) among sub-tasks are labeled with solid lines in the figure. Note that, the solid lines in the figure only capture data reuses, *not* data dependencies. At a high level, our approach leverages data reuse to *cluster* sub-tasks and generates *new* tasks. It is to be noted that, after clustering, the execution order of sub-tasks is subject to *data dependencies*, which are captured using a data dependency graph. We provide a detailed discussion of data dependencies in Section 5. Figure 4(b) illustrates the newly-generated tasks based on the data reuse among sub-tasks. For example, the new $Task - 1$ now consists of $T11$, $T21$, $T22$, and $T23$, as these four sub-tasks have data reuse among them as shown in Figure 4(a). Similarly, other sub-tasks are clustered and new tasks are generated based on the data reuse information. As a result, these newly-generated tasks will be scheduled to different cores such that sub-tasks with data reuse among them will be executed on a single core to take advantage of the local cache hierarchy. Note

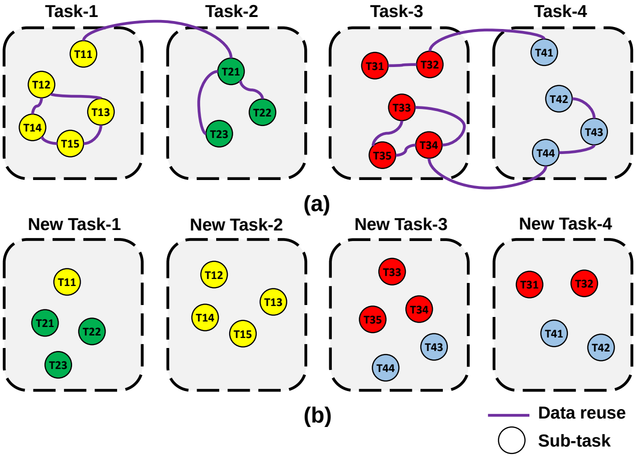


Fig. 4. Data reuse-aware new task generation (i.e., sub-task clustering). (a) Original task parallelization where each cycle represents a sub-task and the data reuses are labeled with solid lines. (b) Newly-formed tasks where sub-tasks with inherent data sharing are clustered together.

that, the number of newly generated tasks is not necessarily identical to the original number of tasks.

Thus, our approach is based on creating new tasks from old tasks in the program, by i) breaking the old tasks into sub-tasks, ii) re-grouping these sub-tasks, and iii) re-ordering them within their new groups.

There are two important points we want to emphasize. First, the number of newly-generated tasks is not necessarily identical to the number of original tasks (though they are the same in this specific example). In our approach, the number of newly-generated tasks is decided by the number of threads specified in the program (i.e., programmer indicates the number of new tasks). Second, when forming the new tasks, our approach also considers "load balancing" across tasks. For instance, in Figure 4(a), there is a group of six sub-tasks ($T33$, $T34$, $T35$, $T42$, $T43$, and $T44$) sharing data and another group of three sub-tasks ($T31$, $T32$, and $T41$) sharing data. To ensure proper load balancing, $T42$ is clustered with $T31$, $T32$, and $T41$ in the new task, instead of having it clustered with the original five sub-tasks which it shares data. Second, for sub-tasks within an original task, it is possible that there is no dependencies between two sub-tasks. As a result, those sub-tasks without dependencies can execute in parallel. However, if there is a data flow between two sub-tasks across the original task boundary, synchronization is required to ensure execution correctness. In our approach, we re-generate the tasks with the goal of reducing the cumulative data movement. Specifically, in the example shown in Figure 4, sub-tasks with data flow are re-organized into new tasks. It is to be emphasized that two-fold benefits are achieved by regenerating the tasks. First, the sub-tasks with intensive data reuse are coalesced into a single task and executed within a particular core to benefit from the local cache hierarchy. Second, the total number of synchronization primitives across tasks is reduced as dependent sub-tasks are grouped within the same new task. That is, while our approach in this paper mainly targets reducing data movement and enhancing data locality, it is also expected to improve the degree of parallelism because we split tasks into smaller granularity (i.e., sub-tasks) and the degree of parallelism across sub-tasks is in general higher than the degree of parallelism across original tasks.

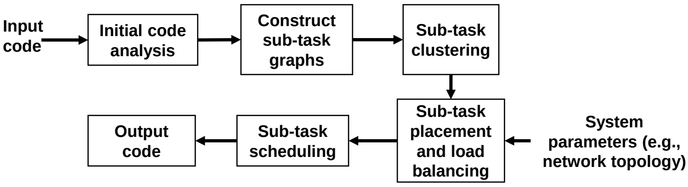

Input code
Initial code analysis
Construct sub-task graphs
Sub-task clustering
Sub-task placement and load balancing
System parameters (e.g., network topology)
Sub-task scheduling
Output code

Fig. 5. Compiler steps to form new tasks in our approach.

5 DETAILS OF OUR APPROACH

Our approach consists of four complementary steps: i) constructing a sub-task graph, ii) sub-task clustering, iii) sub-task to core assignment, and iv) sub-task scheduling. Figure 5 depicts the details of the compiler pass that generates the output code with optimized sub-task assignment and scheduling. Starting with the original input task-parallel code and the user-specified sub-task granularity, we first perform an initial code analysis to construct a *sub-task dependency graph* and *sub-task reuse graph* (Section 5.1), two data structures internal to the compiler. Based on this reuse and dependency information, next we perform sub-task clustering such that the sub-tasks with data reuse among them are clustered into the same new task (Section 5.2). Finally, we perform sub-task assignment and scheduling using the data location information (Section 5.3 and Section 5.4). The final output code also considers load balance across different cores. Algorithm 1 shows the pseudo-code of the compiler algorithm. It takes the number of threads, number of tasks, on-chip network topology, and the user-defined sub-task granularity as inputs, and generates the "sub-task to core assignment and task scheduling" with the goal of maximizing data locality and minimizing data movement on the on-chip network. The complexity of the algorithm is $O(log(N) \times n^2 log(n))$, where N is the number of threads specified in the program and n is the total number of vertices (sub-tasks) in the sub-task graph.

We want to emphasize that, this approach is flexible enough in that it can work with any sub-task granularity. For example, a sub-task can be a single program statement, or a consecutive set of program statements, or a few iterations of a parallel loop, or an entire loop nest. In this paper, given an input application code, we focus on its affine portions where the data access patterns can be determined ahead of execution. For non-affine program portions (e.g., loop nest with indirect data accesses), we do not apply our optimization.

The following subsections discuss the details of our approach.

5.1 Constructing Sub-task Graph

Our approach takes a task graph as input. In the task graph, each node represents a task and an edge between two nodes represents a (control or data) dependency between the corresponding two tasks. The first step in Algorithm 1 is to generate the sub-task graph (line 30). More specifically, given a task graph $TG(V, E)$, where V represents the number of tasks in TG and E represents the number of edges in TG, function `Construct_sub-task_graph` generates two graphs: i) a sub-task data reuse graph G, and ii) a sub-task dependency graph D. In the sub-task data reuse graph, each node represents a sub-task in the granularity of S operations (S being the sub-task size), each edge represents the data reuse between two sub-tasks, and an edge weight captures the "intensity" of data reuse between the sub-tasks. Currently, we rely on the programmer to specify S. We also report sensitivity study results with different sub-task sizes later in our experimental analysis. It is important to emphasize that the sub-task size S affects the data locality and the degree of parallelism of the newly-generated tasks in our approach. Specifically, a large S indicates that more

Algorithm 1 Sub-task clustering, placement, and scheduling

INPUT: Number of threads (N); Number of Tasks (T); Sub-task granularity (S); Task graph TG
OUTPUT: Sub-tasks to core scheduling.

```
//generate sub-task graph and sub-task dependence graph
function CONSTRUCT_SUB-TASK_GRAPH(Task TG(V,E))
    for each task do
        partition each task into sub-task $S_1$, $s_2 \cdot s_k$ with size of $S$ statements
        if $s_i$ has data sharing with $s_j$ then
            add edge from $s_i$ to $s_j$ in sub-task graph $G$
            set the edge weight in $G$ as the intensity of data sharing
        if $s_i$ has data dependence with $s_j$ then
            add arrow from $s_i$ to $s_j$ in sub-task dependence graph $D$
    return $(G, D)$
//Kernighan-Lin algorithm
function GRAPH_PARTITION(sub-task graph G(V,E))
    //randomly partition sub-task graph into $G_a$ and $G_b$ where size($G_a$) = size($G_b$)
    $g_max = 0$
    while $g_max > 0$ do
        for each sub-task $s_i$ in $G_a$ and $s_j$ in $G_b$ do
            Let $WI$ and $WE$ denote the internal cost and external cost of weighted edges, respectively.
            $D_i$ = $WE_i$ - $WI_i$ and $D_j$ = $WE_j$ - $WI_j$
        $G_1$, $G_2$, and $G_m \longleftarrow \varnothing$
        for $i$ from 1 to V/2 do
            find $s_i$ in $G_a$ and $s_j$ in $G_b$ such that $g = D_i + D_j$ -2*c($s_i$,$s_j$) is max.
            add $s_i$ to $G_1$, $s_j$ to $G_2$, and $g$ to $G_{temp}$
            remove $s_i$ from $G_a$ and $s_j$ from $G_b$
            update the $WE$ and $WI$ for the remaining vertices in $G_a$ and $G_b$
        // select k $g$ from $G_{temp}$ such that the sum of k $g$ 9s maximized
        if sum($g_1 \cdot g_k$) > 0 then
            exchange $s_1$, $s_2 \cdot s_k$ from $G_1$ with $s_1$, $s_2 \cdot s_k$ from $G_2$
    return $(G_1, G_2)$
//Step 1: constructing sub-task graph
($G$, $D$) = CONSTRUCT_SUB-TASK_GRAPH(Task $TG$)
//Step 2: clustering sub-tasks (Kernighan-Lin algorithm)
count = 1
while count <= N do
    ($G_a$, $G_b$) = GRAPH_PARTITION(sub-task graph G(V,E))
    $count \longleftarrow 2 * count$
    recursively call GRAPH_PARTITION($G_a$) and GRAPH_PARTITION($G_b$)
//Step 3: sub-task group placement
construct reuse graph among sub-task groups
Calculate weights of the group reuse graph
Place high-reuse groups to neighboring cores.
```

```
41: //Step 4: sub-task scheduling
42: for each core $c_i$ do
43:     for each sub-task $s_{ij}$ assigned to $c_i$ do
44:         for the two neighboring cores $c_r$ and $c_b$ (right and bottom) of $c_i$ do
45:             Choose the sub-tasks $s_{ir}$ and $s_{ib}$ for $c_r$ and $c_b$ such that they have highest reuse
    with $s_{ij}$.
```

data locality can be captured within a sub-task. However, a large S diminishes the effectiveness of sub-task clustering (due to dependencies) and may also lead to imbalanced execution among cores. In contrast, a small S allows more effective sub-task clustering to maximize the parallelism but may distort data locality if the reuses are separated in different sub-tasks. Note that the data reuse captured in the data reuse graph is "dependency-free". In other words, data reuse graph does *not* capture the data dependencies between sub-tasks, meaning that any subset of sub-tasks from the data reuse graph can potentially execute in parallel. Instead, the dependency information among sub-tasks is captured separately in the sub-task dependency graph D, the second data structure our compiler employs. We want to emphasize two important points regarding the sub-task dependency graph. First, the sub-task dependency graph is *different* from the original task-level dependency graph. To further explain this point, let us consider an example of two dependent tasks $T1$ and $T2$ ($T1 \longrightarrow T2$) in the original task graph. For explanation purposes, let us further assume that $T1$ is partitioned into two sub-tasks $T11$ and $T12$, and $T2$ is also partitioned into two sub-tasks $T21$ and $T22$. Supposing that the data dependency in the original task graph ($T1 \longrightarrow T2$) is between two statements $S1$ and $S2$, and after task partitioning, $S1$ resides in $T11$ whereas $S2$ resides in $T22$, then the dependency is captured by an edge from $T11$ to $T22$ ($T11 \longrightarrow T22$). As a result, one potential benefit of our approach is that task partitioning can possibly generate "more parallelism" compared to the original task graph, i.e., the degree of parallelism can increase. Specifically, in the aforementioned example, $T21$ can execute in parallel with $T11$ and $T12$ as it does not depend on $T11$ and $T12$. The second important point regarding the sub-task graph is that, we need to capture the "newly-generated" dependencies. For example, if $T12$ consumes the data generated by $T11$, an edge from $T11$ to $T12$ ($T11 \longrightarrow T12$) is inserted in the sub-task dependency graph.

5.2 Sub-task Clustering

The next step is to generate "new tasks" that are eventually mapped to cores. The number of new tasks is determined by the number of threads specified in the program (N).[2] Our goal is to partition the sub-task graph into N groups (new tasks), and assign these new tasks to different cores, such that the data reuse within a new task is maximized and data reuse across tasks is minimized. To this end, we recursively apply the Kernighan-Lin algorithm [18] (lines 32 to 36) to the sub-task reuse graph. Specifically, Kernighan-Lin algorithm partitions the sub-task graph G into two sub-graphs $G1$ and $G2$ with equal sizes (i.e.,the same number of sub-tasks). It guarantees that the cumulative weights of the edges that cross the $G1$-$G2$ boundary is minimized. We then apply the same process to $G1$ and $G2$ separately, until the number of sub-graphs is equal to the number of threads specified in the program. In a sense, these sub-graphs are the new tasks. Note that load balancing is automatically taken care of, as the algorithm partitions the sub-task graph into same-sized sub-graphs (i.e., new tasks).

After generating new tasks, we insert *synchronization primitives* to ensure execution correctness. Specifically, if two dependent sub-tasks (observed from the sub-task dependency graph) reside in two different newly-generated tasks, we insert synchronization before the dependent sub-task.

[2] As stated earlier, this can be set to the number of cores or any other value.

This is because, different tasks are assigned to different cores and they can execute in parallel. Note that, if the two dependent sub-tasks reside in the same task, we do *not* need any synchronization as they execute sequentially on the same core.

5.3 Sub-task Placement

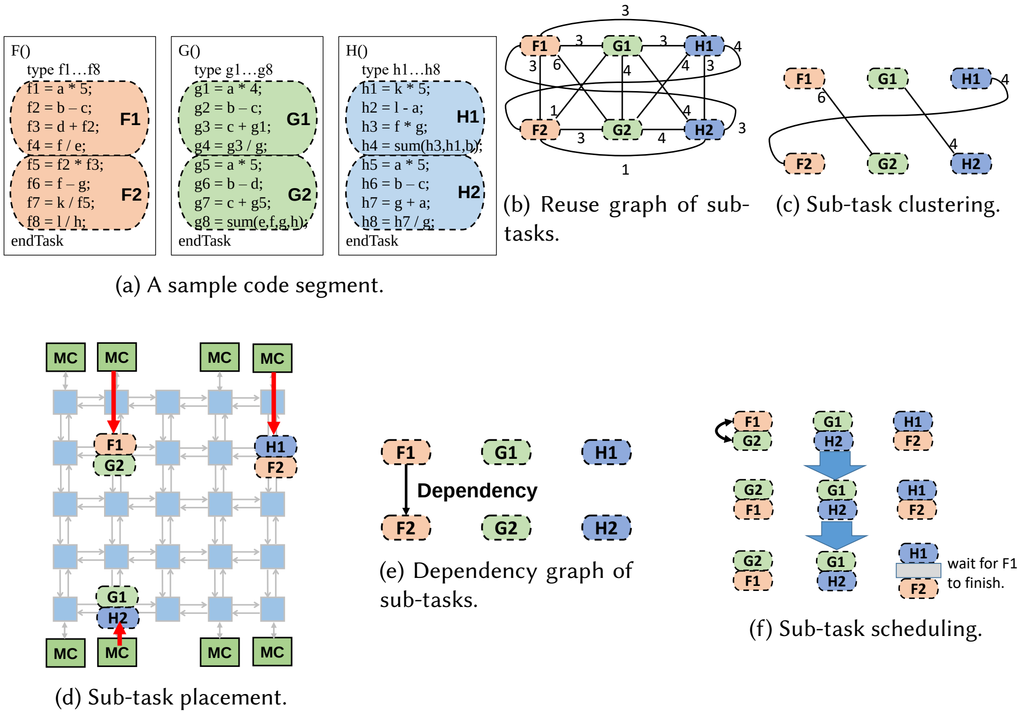
The next step is to assign the newly-generated tasks to cores. Recall from the discussion in Section 2 that, multiple cores are connected through a 2D mesh-based on-chip network in our targeted manycore architecture. Therefore, if the requested data can be found in the local cache hierarchy or can be supplied from the nearby cores, the incurred data access latency would be lower, compared to the case where the requested data is supplied from the remote nodes or fetched from the off-chip memory. In this paper, we leverage this observation, by generating a reuse graph among all newly-generated tasks (lines 38 to 40). The weights on the edges capture the "intensity" of data reuse among new tasks. Note that, although most of the inherent data sharing is wrapped within new tasks through our sub-task clustering, there will still exist data sharing among newly-generated tasks. Our task-level reuse graph captures such opportunities and exploits them in assigning the new tasks to cores. More specifically, every time we assign a task to core, we choose the one that has most data reuse (i.e., larger weights on the edges) with other tasks that have already been assigned to the nearby cores. That is, the goal is to have the new tasks with intense data sharing getting assigned to nearby cores such that the total amount of data movement on the on-chip network is minimized as much as possible.

5.4 Sub-task Scheduling

The last step in our approach is to reorder/schedule the sub-tasks within new tasks such that the sub-tasks with inherent data sharing across tasks/cores execute in parallel or closer in time (lines 42 to 45). For instance, recall our example in Figure 3c where, after sub-task clustering, sub-task $F-1$ accesses data structure c at the beginning of the new task (i.e., the new task includes $F-1$ and $G-2$), whereas sub-task $F-2$ accesses c towards the end of the new task (i.e., the new task includes $G-1$ and $F-2$). The problem is that, by the time $F-2$ requests c, it is likely that c is no longer in the cache of the node where $F-1$ executes. In other words, the reuse of data structure c between $F-1$ and $F-2$ would not be converted into data locality due to the fact that there is a large time interval between the executions of $F-1$ and $F-2$. Our sub-task scheduling fixes this problem by shuffling the execution order of sub-tasks within each newly-formed task. Note that our scheduling algorithm is essentially a greedy heuristic, in that we first choose one task and then shuffle the sub-tasks in other tasks based on the reuse information from the sub-task reuse graph. Further, sub-task reordering should not violate the dependencies among sub-tasks. We check the sub-task dependency graph before each reordering to ensure the correctness of execution.

5.5 Example

To help better explain the operations of our approach, let us consider the example shown in Figure 6. Figure 6a shows an example program fragment with three tasks: $F()$, $G()$, and $H()$. Each task consists of eight statements, and we assume a sub-task granularity of four statements. Regarding the data structures, each task has some local/private data structures (e.g., from $f1$ to $f8$) as well as some shared data structures (e.g., a and b). Since the granularity of sub-tasks is four statements, each of the original tasks is divided into two sub-tasks. For example, $F()$ is divided into two sub-tasks labeled with $F1$ and $F2$. Figure 6b shows the data reuse graph built from these sub-tasks, and Figure 6e depicts the sub-task dependency graph after executing the first step of Algorithm 1. Note that the data reuse graph is a complete graph with edge weights representing the intensity of data sharing among sub-tasks. Let us further assume that the number of new tasks formed is three.



```
F()
    type f1…f8
    f1 = a * 5;
    f2 = b – c;
    f3 = d + f2;
    f4 = f / e;
    f5 = f2 * f3;
    f6 = f – g;
    f7 = k / f5;
    f8 = l / h;
endTask
```

```
G()
    type g1…g8
    g1 = a * 4;
    g2 = b – c;
    g3 = c + g1;
    g4 = g3 / g;
    g5 = a * 5;
    g6 = b – d;
    g7 = c + g5;
    g8 = sum(e,f,g,h);
endTask
```

```
H()
    type h1…h8
    h1 = k * 5;
    h2 = l - a;
    h3 = f * g;
    h4 = sum(h3,h1,b);
    h5 = a * 5;
    h6 = b – c;
    h7 = g + a;
    h8 = h7 / g;
endTask
```

(a) A sample code segment.

(b) Reuse graph of sub-tasks.

(c) Sub-task clustering.

(d) Sub-task placement.

(e) Dependency graph of sub-tasks.

(f) Sub-task scheduling.

Fig. 6. An example to explain the steps of our proposed approach.

Figure 6c shows the generated new tasks after the second step of Algorithm 1. Specifically, sub-tasks $F1$ and $G2$ are clustered into a new task. Similarly, ($G1$ and $H2$) and ($H1$ and $F2$) are clustered into another two separate new tasks. Finally, these three newly-generated tasks are assigned to three different cores for execution based on the data location (i.e., distance to the memory controllers) of their requested data (Figure 6d). It is important to emphasize that, since there is a data dependency from $F1$ to $F2$ (Figure 6e), a synchronization primitive is inserted before the execution of $F2$ to ensure that $F2$ is executed *after* $F1$ finishes execution in a different core (Figure 6f). Figure 6f also shows sub-task scheduling within a newly-generated task, which is the last step of our algorithm. The execution orders of $F1$ and $G2$ are switched because $G2$ has data reuse with $H1$. After the scheduling, $G2$ and $H1$ are executed in parallel (or close in time) so that the reused data has a higher chance of residing in the on-chip cache at the time of reuse (i.e., higher chance of getting converted into data locality). The final output of our algorithm is shown in the bottom of Figure 6f.

6 EXPERIMENTAL EVALUATION

In this section, we first describe our experimental framework and benchmark programs, and then present our detailed experimental evaluation. In our evaluations, we compare our proposed approach against a baseline as well as two recently-published approaches (one compiler-based and one hardware-driven).

6.1 Experimental Setup

Target Architecture. As stated earlier, we used an Intel Xeon Phi based architecture (Knight's Landing – KNL) to test the effectiveness of our proposed optimization approach. Most of our experiments have been performed when the architecture is configured in the cache mode with the

Table 1. Compile-time statistics.

Benchmark Name	Number of of Tasks	Number of Sub-tasks	Task Diversity
barnes	12	8,184	6.5
fmm	18	10,052	11.2
radiosity	24	12,392	14.1
raytrace	8	9,973	5.5
volrend	12	14,833	7.8
water	20	6,891	8.3
cholesky	10	7,152	6.1
fft	16	16,927	10.4
lu	8	8,244	5.8
radix	16	8,707	11.0

sub-NUMA cluster mode, as the baseline version (explained shortly) generated the best results with this setup. This is because in the sub-NUMA cluster mode, both the L2 data accesses and memory accesses are limited within a sub-NUMA region, and as a result, the worst-case data fetching path is shorter compared to other two modes. However, later we also present results with cache mode + quadrant mode.

Compiler Infrastructure and Runtime Environment. We used LLVM version 8.0 [31], to implement our proposed optimization as a “compiler pass”. In the resulting compiler, which provides a state-of-the-art, SSA-based compilation strategy, our pass is invoked (to partition computations across cores) before data locality optimizations, vector parallelism optimizations, and low-level (core-level) optimizations have been performed. We noticed that our added compiler pass increased average compilation time by 36%, resulting in a maximum compilation time of 13.1 seconds (across all benchmark programs tested). In our experiments, the machine was running Linux Kernel 4.14.

Versions. In this work, we have used four different versions explained below. All versions take the original code as input but they differ how computations are assigned to cores.

- **Baseline**: This is the default version, which uses/obeys original (pThreads/OpenMP based) parallelization directives. In this case, each core performs a separate task by working on a completely different function (task parallelism), or by executing a subset of the iterations of a parallel loop nest (data parallelism). Dependencies among tasks are followed per our discussion in Section 3. Unless otherwise stated, the results with the remaining versions are *normalized* with respect to this version. Note that this version already uses all the optimizations the Intel compiler (*icc*) offers.
- **M&M**: This represents the optimization strategy (mix-and-match) presented in this paper. As already discussed in detail, it is based on re-shuffling the placement and scheduling of computations (sub-tasks) across cores.
- **Alt-1** [26]: This is a recently-published compiler-based optimization strategy, which takes into account the relative positions of cores, last-level caches (LLCs), and memory controllers in the target architecture, and generates a mapping of loop iterations to cores, with the goal of minimizing the on-chip network traffic.
- **Alt-2** [16]: This is a hardware-driven application-to-core mapping strategy that targets multiprogrammed workloads and reduces inter-application interference in the on-chip network and memory controllers. Specifically, it i) maps latency sensitive applications to separate parts of the network from bandwidth intensive applications to reduce interference, and ii) maps those applications that benefit more from being closer to the memory controllers close to these resources.

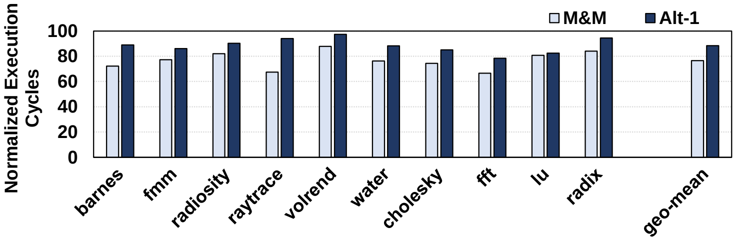


Fig. 7. Execution time results.

We want to mention that, while Alt-1 targets *multithreaded* workloads, Alt-2 targets *multiprogrammed* workloads. Therefore, when we execute a single multithreaded application program, we compare our approach against Alt-1, whereas when we execute multiple (multithreaded) applications, we compare our approach against Alt-2. Note that, our approach is not simply a "superset" of Alt-1 and Alt-2. Rather, it is a general framework that works for both multi-threaded and multi-programmed execution scenarios. Specifically, unlike Alt-1 which focuses on loops and employs loop iteration as the granularity of computation scheduling, our approach focuses on tasks that have a larger granularity that provides the compiler with the capability to re-organize, re-order, and re-map tasks for both parallelism and data locality purposes. On the other hand, our approach is different from Alt-2 as well, which focuses on row-buffer hits and off-chip memory locality. Moreover, our approach does not require any hardware modifications. Also, except for the multiprogrammed workload experiments presented later, in all experiments, we execute one multithreaded application at a time, and the application is parallelized over 36 (6×6) threads. In each node (tile) in our architecture, we use only one of the threads. Later, we also report experimental results collected by assuming different number of threads per node.

Benchmark Programs. We performed experiments using applications from three different benchmark suites: Splash-2 [58], SPECOMP [8], and Mantevo [1]. For now, we present the detailed results for only Splash-2 suite; the results for the other two benchmarks are summarized towards the end of this section. We used 10 Splash-2 applications [58] as our benchmark programs, to test the impact of our proposed optimization approach.[3] In an attempt to increase pressure on the cache/memory sub-system, we increased the input sizes of Splash-2 benchmarks, and the resulting input sizes varied between 33.1 MB and 1.4 GB, and the corresponding LLC (L2) cache miss rates ranged between 18.1% and 38.4%, when using the baseline versions. Unless otherwise stated, we use a default sub-task granularity of 1 program statement.

6.2 Results

Compile-Time Statistics. We start by presenting, in Table 1, some statistics (for Splash-2 benchmarks) that demonstrate the working of the compiler. The second and third columns in this table give, respectively, for each benchmark program tested, the total number of tasks and the total number of sub-tasks. The fourth column on the other hand gives, for a newly-formed task, the number of different (original, old) tasks that supplied a sub-task. That is, task diversity is calculated using the number of different original tasks whose sub-tasks are used in forming the newly-formed task. The number in the Table is an average value across all newly-formed tasks. We see that, on average, each new task has received sub-tasks (after our optimization) from 8.6 different (original) tasks, indicating that our approach moves/reshuffles sub-tasks across tasks quite aggressively.

[3] The only remaining application in the Splash-2 suite, *ocean*, could not be compiled in our target platform, due to some incompatibility issue between the compiler and the OS.

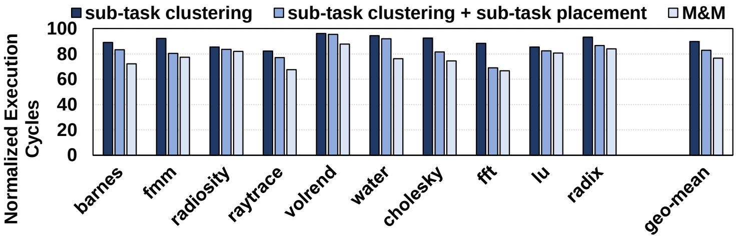


Fig. 8. Incremental evaluation of various steps in M&M.

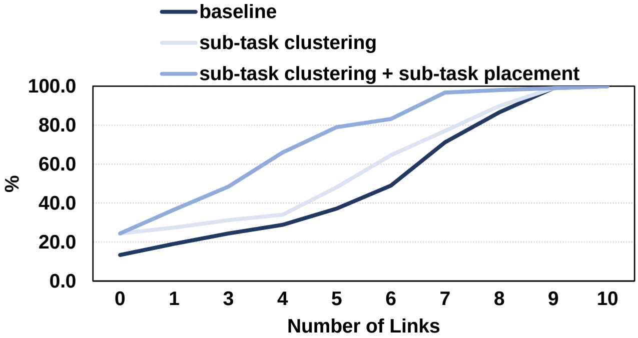


Fig. 9. CDF of Manhattan distances.

Execution Time Results. The graph in Figure 7 gives the execution cycles when using two different schemes (our proposed scheme and Alt-1[4]). All results are *normalized* to the baseline version. We see from these results that our approach improves the performance of all benchmark programs, and the average performance improvement brought by our approach over the baseline version is about 23.4%. It can also be seen that Alt-1 brings an average improvement of 11.7% over the baseline. The main reason why our approach generates higher savings than Alt-1 is because, unlike Alt-1 which is essentially a loop nest-centric optimization strategy, our approach is more general, i.e., it is applicable to any code fragment that can be partitioned into sub-tasks.

Recall that our approach has four steps, namely, sub-task graph formation, sub-task clustering, sub-task placement, and sub-task scheduling. We now quantify the importance of these individual steps. For each benchmark program, the first bar in Figure 8 shows the results (normalized execution times) when only the sub-task graph formation and sub-task clustering steps are used, and remaining two steps (sub-task placement, and sub-task scheduling) are as in the baseline case. The second bar on the other hand represents the results when the sub-task graph formation, sub-task clustering, and sub-task placement of our approach are activated, and the last step is implemented as in the baseline. Finally, the third bar reproduces the results from Figure 7. It can be observed from these results that, each of the sub-task clustering, sub-task placement, and sub-task scheduling steps is crucial for the success of our approach, and that the incremental savings (execution time improvements) brought by them are 10.1%, 6.9% and 6.2%, in that order.

To explain the difference the second and third bars in Figure 8, we also report the Manhattan Distances (MD) of data reuses. The MD of a data reuse is the physical distance (in terms of the number of links) in the on-chip network between the requesting core and the location of the requested data. Figure 9 gives the CDF of MDs, for these two versions. In this graph, a point $(x, y\%)$

[4] Comparison results against Alt-2 will be given later with multiprogrammed workloads (of multithreaded applications).

indicates that $y\%$ of the data accesses traverse x or fewer communication links on the on-chip network. It can be observed from these results that activating the sub-task placement step reduces distance-to-data significantly.

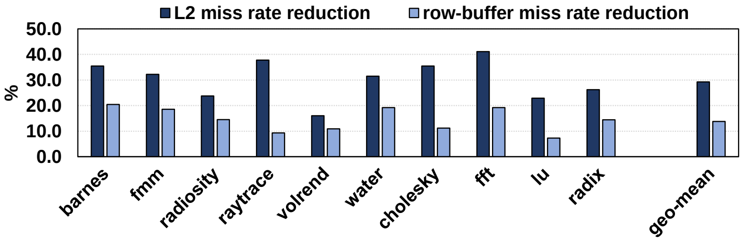


Fig. 10. Reductions in L2 misses and row-buffer misses compared to the baseline.

In addition to this reduction in the on-chip message traffic, our approach also improves on-chip cache performance as well as row-buffer hit rates, primarily due to the improved task locality (i.e., each newly-formed task includes sub-tasks that exhibit "high data reuse" among themselves). In DRAM, memory elements are laid out in arrays of rows and columns, and an access to the memory array occurs at the granularity of a row. The accessed memory row is brought into a buffer, called *row-buffer*, and successive accesses to the same row can be satisfied from the row-buffer, instead of accessing the memory array. Such memory accesses are called row-buffer hits (as opposed to row-buffer misses/conflicts which occur when the accessed memory row does not currently reside in the row-buffer), and a high row-buffer hit rate is usually correlated with high memory performance. Figure 10 presents the improvements (reductions) in the L2 cache misses and row-buffer misses, with respect to the baseline, when our approach is used.[5] It can be observed from the results presented in Figure 10 that the geometric means for cache (L2) miss reduction and row-buffer miss reduction are, respectively, 29.2% and 13.8%.

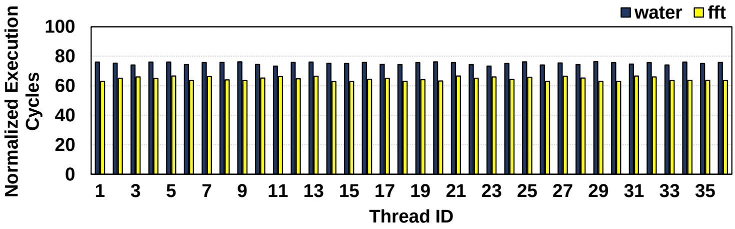


Fig. 11. Normalized execution times of different threads of two of our application programs.

Behavior of Load Balancing. Before we start the discussion of comparison of our approach against optimal (explained shortly), we want to present results that explain the load balance behavior of our approach. As discussed earlier, our approach tries to achieve load balance across tasks, in terms of the sub-tasks they contain. That is, it assigns more or less the same number of sub-tasks to each and every (newly-formed) task. However, the question is whether this sub-task level load balancing also leads to execution time balancing across the different tasks of a given application. We present in Figure 11 the execution times of all 36 threads of two of our benchmark programs,

[5] Since the row-buffer statistics are not possible to collect directly from the system, we estimated them by instrumenting the source code and recording accesses to different memory rows.

water and *fft* (other benchmarks also exhibit similar behavior). It can be seen that the execution times of different tasks are quite balanced, and the difference between the minimum and maximum execution time of any thread pair is 3% for *water* and 3.8% for *fft*.

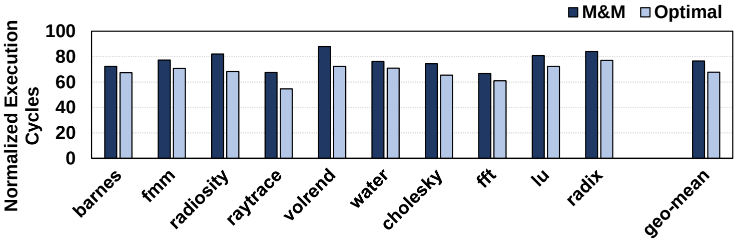


Fig. 12. Comparison against optimal scheme.

Comparison against Optimal. Next, we compare our approach to an optimal scheme. In this context, what we mean by being "optimal" is that the application code is first profiled and our compiler pass is fed with accurate edge weights[6] in the data reuse graph. In other words, the optimal in this case is a version of our proposed approach that operates with accurate edge weight stats. The results plotted in Figure 12 reveal that the average difference between our scheme and optimal is about 8.9%, meaning that a more accurate compiler analysis has scope to further increase the effectiveness of our approach. It can also be noticed that the performance gap between our approach and optimal is larger in graphics programs such as *volrend*, *radiosity* and *raytrace*, primarily because these benchmarks employ mostly unstructured data access patterns.

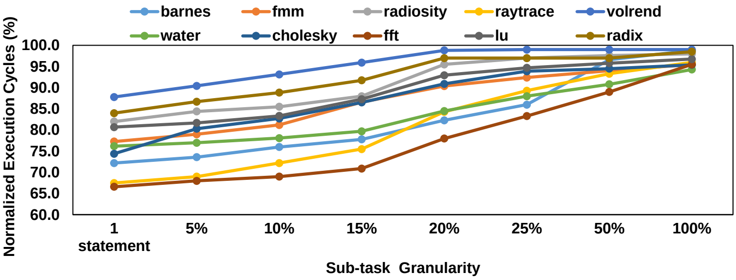


Fig. 13. Results with different sub-task sizes.

Sensitivity to the Sub-task Size. Recall that so far in our experiments we used a sub-task size of 1 program statement. In this part of our evaluation, we vary the sub-task size and measure the sensitivity of our results to it. In Figure 13, a point *a*% on the x-axis indicates a sub-task size which corresponds *a*% of the average task size. Consequently, as we go on the x-axis from left to right, we increase the sub-task granularity. It can be observed from these results that, as long as we work with a sub-task granularity that falls between 1 statement and 15% of the average task size, the resulting performance improvements are consistent. Beyond 15%, the locality starts to get hurt, and we see in some benchmarks significant drop in our savings.

[6] Due to data reuses that cannot be caught by the compiler, e.g., reuses originating from indirect array accesses and pointers.

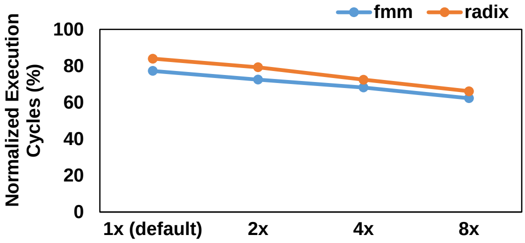


Fig. 14. Results with different dataset sizes.

Results with Increased Dataset Sizes. Next, we study the sensitivity of our savings to the dataset sizes. For this set of experiments, we focus on two of our benchmark programs (*fmm* and *radix*) and increase their dataset sizes from their default values we have been using so far to 2x, 4x and 8x of the default values, and present execution time improvements. The results plotted in Figure 14 indicate that the percentage improvements brought by our approach increase as we increase dataset sizes. This is primarily because a larger input data puts more pressure on cache, memory and on-chip network resources, rendering data access optimization even more critical.

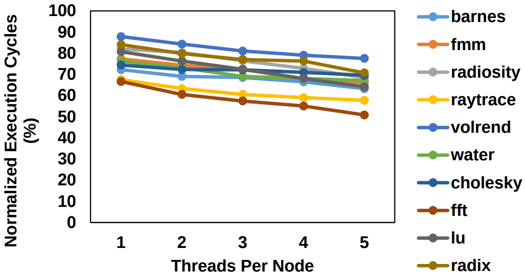


Fig. 15. Sensitivity to the number of threads per node.

Results with Varying Number of Threads Per Node. Recall that so far in our experiments we used 1 thread per node (tile). In this part of our evaluation, we test the effectiveness of our approach with different thread counts per node. The results plotted in Figure 15 indicate that, the percentage improvements brought by our approach increases as the number of threads per node increases. This is primarily because a large number of threads running in parallel put more pressure on caches, on-chip network, and row-buffers. Consequently, improving data locality and reducing on-chip data movement can have a larger impact on performance. On average, we observe 27.1%, 29.7%, 32% and 34.9% performance improvements, with 2, 3, 4 and 5 threads per node, respectively.

Results with Quadrant Cluster Mode. Recall that the results presented so far have been collected under the cache mode with sub-NUMA cluster mode. In this part of our evaluation, we present execution time improvements with the cache mode + quadrant cluster mode. The results plotted in Figure 16[7] clearly show that the relative performances of M&M and Alt-1 are similar to those obtained when using the cache mode + sub-NUMA node.

Results with Multiprogrammed Workloads of Multithreaded Applications. Now, we evaluate the performance of our approach in a multiprogrammed environment and compare

[7] The bars are *normalized* to the baseline when using cache mode + quadrant mode.

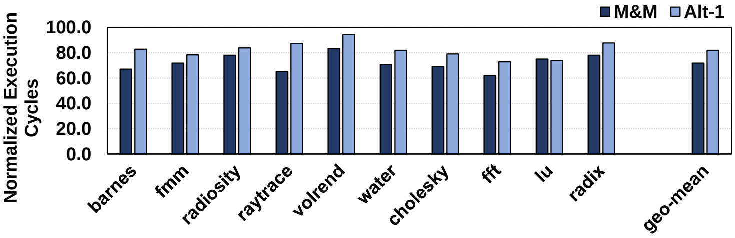


Fig. 16. Results with cache memory mode and quadrant cluster computing mode.

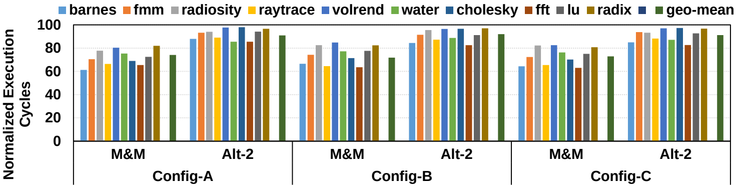


Fig. 17. Comparison of M&M and Alt-2 in three different configurations.

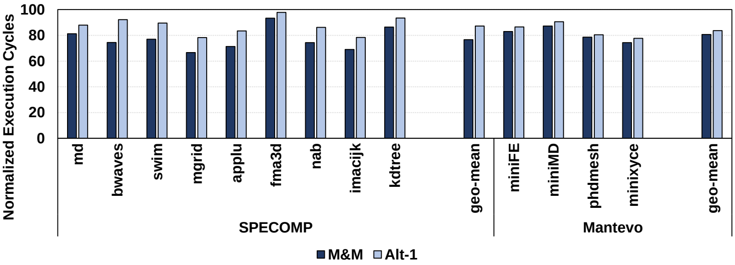


Fig. 18. Normalized execution times with benchmark programs from SPECOMP and Mantevo.

it against an alternate approach (Alt-2). As discussed earlier, Alt-2 is a *hardware-driven* application-to-core mapping strategy that targets multiprogrammed workloads and reduces inter-application interference in the on-chip network and memory controllers. In Figure 17, we compare our approach and Alt-2 in three different configurations. In Config-A, in each experiment, we run two copies of the application being tested; each of the copies is parallelized over 36 threads. As a result, one of the cores in each node runs one thread from one of the application copies, whereas the other core in the same node runs one thread from the other copy. In Config-B, each application is parallelized over 36 threads (as in Config-A) but this time, all threads of one application copy runs on the left half of the architecture and all threads of the other copy runs on the right half. Consequently, in a given node, both the cores run threads from the same copy. Finally, in Config-C, we run four copies of an application, each parallelized over 18 threads which are assigned to a 3x3 sub-grid of the architecture. The results plotted in Figure 17 reveal that i) our approach brings improvements in all three configurations tested, ii) its savings are higher with Config-A, as chances for contention are higher in the baseline execution under Config-A due to more intense interleaving of the data accesses from different application copies, and iii) our approach generates better results than Alt-2 in all configurations and in all benchmarks. This is due to the fact that a) Alt-2 is mostly optimized for multiprogrammed workloads of single-threaded applications, and b) it primarily targets memory locality (row-buffer hit rate), *not* cache performance.

Results with Other Benchmark Suites. Finally, we present the results collected using the application programs from the SPECOMP and Mantevo benchmark suites. Due to space concerns, only execution time results are presented, but (though not presented here) the sensitivity results with SPECOMP and Mantevo exhibit similar trends to the sensitivity results with Splash-2. For these experiments, sub-NUMA mode of KNL is used. It can be observed from Figure 18 that, our approach brings an average execution time reduction of 23.3% with SPECOMP benchmarks and 19.3% with Mantevo benchmarks, and the corresponding performance improvements with Alt-1 are 12.7% and 16.3%, respectively. Furthermore, our approach outperforms Alt-1 in all benchmark programs tested.

Discussion: We want to discuss several aspects of our approach. First, the current implementation relies on programmers to provide the sub-task granularity. This introduces the programmers' burden and may shift the benefits one can obtain from our approach due to an improper sub-task size. Second, the sub-task size is fixed, and as a result, certain parallelism or data locality may be sacrificed due to the fixed size. We will explore compiler dynamic sub-task size determination in our future work. Finally, our scheduling policy is a greedy-based algorithm. As a result, it may not achieve optimal scheduling, and consequently there can be opportunities missed by our approach.

7 RELATED WORK

In this section, we discuss prior efforts related to our work from three aspects: i) conventional data locality optimizations, ii) near-data computing, and iii) recomputation with near data.

Conventional data locality optimizations: Conventional data locality optimizations strive to improve hit rate in cache hierarchy and reduce the amount of data requests reaching lower level caches and memory system [6, 14, 17, 20, 23, 27, 30, 32, 34, 46, 47, 51, 52, 54, 55]. Sundararajah et al. [48] targeted nested recursions in programs, and proposed a set of scheduling transformations to automatically improves locality at all levels of memory hierarchy. Focusing on GPUs, Li et al. [33] characterized the data locality opportunities in GPUs and proposed thread block scheduling policies which are data locality aware. Chen et al. [13] introduced an approach of locality analysis based on static parallel sampling. Specifically, a compiler analysis pass generates sampler codes to measure the locality, and then, the precise locality is inferred from the execution of sampler code. Our approach introduced in this paper can be considered as complementary to most of these prior efforts on cache miss minimization. Targeting task-parallel applications, our idea is to organize and schedule tasks that are data locality aware. Therefore, the conventional optimizations (e.g., loop transformation) can be applied to computations within tasks to further improve the data access performance.

Near-data computing: One popular execution paradigm which targets reducing the cost of data movement is near-data computing (NDC) or near-data processing (NDP). Instead of fetching data to computation, NDC offloads computation to near data and performs computation in places where the requested data are nearby. Prior works explored the opportunities of employing the concept of NDC from different aspects including compiler optimizations, operating systems support, and architectural optimizations [3, 4, 9, 10, 12, 21, 28, 36, 38, 44, 60]. Ahn et al. [3] proposed a processing-in-memory (PIM) architecture together with PIM instructions and dynamically scheduled the PIM instructions on processor or in-memory based on data locality. Chu et al. [15] developed high-level programming extensions to support PIM-aware programming. Their approach allows programmers to control the applications running on PIM architectures. Pattnaik et al. [41] proposed learning-based mechanisms to schedule computations in NDC-based GPU architectures. Hsieh et al. [21] implemented schemes that offload selected code segments to execute in memory without any need of program modification. Compared to all these prior NDC-centric works, our approach not only schedules computations (i.e., sub-tasks) to cores in a data reuse-aware fashion, but also changes the computation execution order such that the intrinsic data reuses across different computations are captured by on-chip caches. Unlike the NDC-based approaches, instead of scheduling computations near data for execution, we reorganize computations to have them executed when the required data are nearby. Consequently, unlike the NDC approaches, our strategy does not require any special architectural support, and consequently, it can be used with any manycore architecture.

Recomputation with near data: There are several prior research efforts focusing on recomputing the requested data using nearby data to reduce the on-chip network data movement. Tang et al. [49] proposed a software approach where a costly data access in a program slice is replaced by a few less costly data accesses plus some extra computations if this replacement reduces the total cost. Their approach requires heavy program analysis to guarantee the correctness and efficiency of generated program with recomputation. Akturk and Karpuzcu [5] proposed an energy-efficient architecture-based recomputation strategy to reduce the pressure on memory hierarchy and communication bandwidth. Their approach requires additional hardware to hold and schedule recomputation slices (*Rslice* in their paper). Adams et al. [2] focused on cloud computing and proposed a cost model to analyze the trade-offs between storing the provenance data or recomputing the data. Compared to all these prior efforts, our approach does *not* use recomputation to reduce data movement. Instead, we focus on task graph-based applications, and we divide tasks into *sub-tasks* and schedule sub-tasks into nearby locations to reduce data movement. Specifically, the advantage of our approach is four-fold. First, it effectively reduces the total data movement over the on-chip network without any need of hardware modification and extensive program code analysis. Second, it introduces additional parallelism compared to original task execution. Third, it improves data locality and cache performance through sub-task scheduling. And finally, unlike recomputation, our approach is applicable to a broader set of application programs.

8 CONCLUDING REMARKS AND FUTURE WORK

The main contribution of this paper is a new compiler algorithm that minimizes data movement on the on-chip network of manycore systems. An important characteristic of this algorithm is that it can optimize any application program that can be represented by a task graph. As such, it is more general than many previously-proposed data locality optimization strategies. We implemented this compiler algorithm in LLVM and performed experiments using different multithreaded benchmark programs on a state-of-the-art manycore system. Our experimental evaluations indicate that the proposed compiler algorithm i) improves the performance of 10 multithreaded programs tested (23.4% on average), ii) outperforms two previously-proposed state-of-the-art data accesses optimization strategies (one compiler and one hardware based), iii) is effective with multiprogrammed workloads as well, and iv) generates results that are close to the maximum savings that could be achieved with perfect profile information.

Our current work includes testing this data movement minimization algorithm on cloud workloads and other types of manycore architectures. In parallel, we also explore how to integrate our approach with conventional loop parallelization techniques, and whether it can be made to be part of a runtime system that employs dynamic compilation.

ACKNOWLEDGMENTS

The authors thank Dr. Evgenia Smirni for shepherding the paper. The authors would also like to thank the anonymous SIGMETRICS reviewers for their constructive feedback and suggestions.

This work is supported in part by NSF grants #1908793, #1629915, #1629129, #1763681, #2028929, #2008398, #2011146, and #1931531, as well as a startup funding from the University of Pittsburgh.

REFERENCES

[1] 2012. minighost. https://mantevo.org/default.php.

[2] Ian F. Adams, Darrell D. E. Long, Ethan L. Miller, Shankar Pasupathy, and Mark W. Storer. 2009. Maximizing Efficiency by Trading Storage for Computation. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*.

[3] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*.

[4] Berkin Akin, Franz Franchetti, and James C. Hoe. 2015. Data Reorganization in Memory Using 3D-stacked DRAM. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*.

[5] Ismail Akturk and Ulya R. Karpuzcu. 2017. AMNESIAC: Amnesic Automatic Computer. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*.

[6] Jennifer M. Anderson and Monica S. Lam. 1993. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[7] Alexandr Andoni, Huy L. Nguyen, Aleksandar Nikolov, Ilya Razenshteyn, and Erik Waingarten. 2017. Approximate Near Neighbors for General Symmetric Norms. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing (STOC 2017)*.

[8] Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. 2001. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *OpenMP Shared Memory Parallel Programming*, Rudolf Eigenmann and Michael J. Voss (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–10.

[9] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. 2014. Near-data processing: Insights from a MICRO-46 Workshop. *Micro, IEEE* (2014).

[10] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. 2017. It's time to think about an operating system for near data processing architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. 56–61.

[11] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. 1994. Compiler Optimizations for Improving Data Locality. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[12] J. Carter, W. Hsieh, L. Stoller, M. Swanson, Lixin Zhang, E. Brunvand, A. Davis, Chen-Chi Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. 1999. Impulse: Building a Smarter Memory Controller. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture (ISCA)*.

[13] Dong Chen, Fangzhou Liu, Chen Ding, and Sreepathi Pai. 2018. Locality Analysis Through Static Parallel Sampling. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[14] Dongjoo Choi, Myunghoon Jeon, Namgi Kim, and Byoung-Dai Lee. 2017. An enhanced data-locality-aware task scheduling algorithm for hadoop applications. *IEEE Systems Journal* 12, 4 (2017), 3346–3357.

[15] Michael L. Chu, Nuwan Jayasena, Dong Ping Zhang, and Mike Ignatowski. 2013. High-level Programming Model Abstractions for Processing in Memory. *Proc. of 1st Workshop on Near-Data Processing in conjunction with MICRO* (2013).

[16] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi. 2013. Application-to-core mapping policies to reduce memory system interference in multi-core systems. In *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*.

[17] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *ASPLOS*.

[18] Shantanu Dutt. 1993. New Faster Kernighan-Lin-type Graph-partitioning Algorithms. In *Proceedings of the IEEE/ACM International Conference on Computer-aided Design (ICCAD)*.

[19] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Biscuit: A Framework for Near-data Processing of Big Data Workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*.

[20] Mary H. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. 1995. Detecting Coarse-grain Parallelism Using an Interprocedural Parallelizing Compiler. In *Supercomputing*.

[21] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. 2016. Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent

Near-Data Processing in GPU Systems. In *ISCA*.

[22] Mahmut Kandemir, Alok Choudhary, J Ramanujam, and Prith Banerjee. 1999. A matrix-based approach to global locality optimization. *J. Parallel and Distrib. Comput.* (1999).

[23] M. Kandemir, J. Ramanujam, A. Choudhary, and P. Banerjee. 2001. A layout-conscious iteration space transformation technique. *IEEE Trans. Comput.* (2001).

[24] Gwangsun Kim, John Kim, Jung Ho Ahn, and Yongkee Kwon. 2014. Memory Network: Enabling Technology for Scalable Near-Data Computing. In *Proceedings of the 2nd Workshop on Near-Data Processing.*

[25] Nam Sung Kim and Pankaj Mehra. 2019. Practical Near-Data Processing to Evolve Memory and Storage Devices into Mainstream Heterogeneous Computing Systems. In *Proceedings of the 56th Annual Design Automation Conference 2019 (DAC '19).*

[26] Orhan Kislal, Jagadish Kotra, Xulong Tang, Mahmut Taylan Kandemir, and Myoungsoo Jung. 2018. Enhancing Computation-to-core Assignment with Physical Location Information. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[27] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. 1997. Data-centric Multi-level Blocking. In *PLDI.*

[28] Jagadish B Kotra, Diana Guttman, Mahmut T Kandemir, Chita R Das, et al. 2017. Quantifying the potential benefits of on-chip near-data computing in manycore processors. In *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS).* IEEE, 198–209.

[29] J. B. Kotra, D. Guttman, N. C. N., M. T. Kandemir, and C. R. Das. 2017. Quantifying the Potential Benefits of On-chip Near-Data Computing in Manycore Processors. In *25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS).*

[30] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. 1991. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[31] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO).*

[32] Shun-Tak Leung and John Zahorjan. 1995. *Optimizing data locality by array restructuring.* Department of Computer Science and Engineering, University of Washington.

[33] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. 2017. Locality-Aware CTA Clustering for Modern GPUs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[34] Amy W. Lim, Gerald I. Cheong, and Monica S. Lam. 1999. An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication. In *ICS.*

[35] Qingda Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Yongjian Chen, Haibo Lin, and Tin-Fook Ngai. 2009. Data Layout Transformation for Enhancing Data Locality on NUCA Chip Multiprocessors. In *Proceedings of the Parallel Architectures and Compilation Techniques (PACT).*

[36] Kyoji Mizoguchi, Shohei Kotaki, Yoshiaki Deguchi, and Ken Takeuchi. 2017. Lateral charge migration suppression of 3D-NAND flash by vth nearing for near data computing. In *2017 IEEE International Electron Devices Meeting (IEDM).* IEEE, 19–2.

[37] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[38] R Nair, SF Antao, C Bertolli, P Bose, JR Brunheroto, T Chen, C-Y Cher, CHA Costa, C Evangelinos, and BM Fleischer. 2015. Active Memory Cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development* (2015).

[39] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[40] OpenMP Architecture Review Board. 2011. *OpenMP Application Program Interface.* Specification. http://www.openmp.org/mp-documents/OpenMP3.1.pdf

[41] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, and Chita R. Das. 2016. Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT).*

[42] Ashutosh Pattnaik, Xulong Tang, Onur Kayiran, Adwait Jog, Asit Mishra, Mahmut T. Kandemir, Anand Sivasubramaniam, and Chita R. Das. 2019. Opportunistic Computing in GPU Architectures. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA).*

[43] Arch D Robison. 2012. Cilk plus: Language support for thread and vector parallelism. *Talk at HP-CAST* 18 (2012), 25.

[44] Gagandeep Singh, Lorenzo Chelini, Stefano Corda, Ahsan Javed Awan, Sander Stuijk, Roel Jordans, Henk Corporaal, and Albert-Jan Boonstra. 2018. A review of near-memory computing architectures: Opportunities and challenges. In *2018 21st Euromicro Conference on Digital System Design (DSD)*. IEEE, 608–617.

[45] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu. 2016. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro* (2016).

[46] Yonghong Song and Zhiyuan Li. 1999. New Tiling Techniques to Improve Cache Temporal Locality. In *PLDI*.

[47] Georgios L Stavrinides and Helen D Karatza. 2020. Orchestration of real-time workflows with varying input data locality in a heterogeneous fog environment. In *2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC)*. IEEE, 202–209.

[48] Kirshanthan Sundararajah, Laith Sakka, and Milind Kulkarni. 2017. Locality Transformations for Nested Recursive Iteration Spaces. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[49] Xulong Tang, Mahmut Taylan Kandemir, Hui Zhao, Myoungsoo Jung, and Mustafa Karakoy. 2019. Computing with Near Data. In *Proceedings of the 2019 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.

[50] Xulong Tang, Orhan Kislal, Mahmut Kandemir, and Mustafa Karakoy. 2017. Data Movement Aware Computation Partitioning. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[51] Xulong Tang, Ashutosh Pattnaik, Onur Kayiran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Das. 2019. Quantifying Data Locality in Dynamic Parallelism in GPUs. *ACM SIGMETRICS Performance Evaluation Review* 47, 1 (2019), 25–26.

[52] Didem Unat, Anshu Dubey, Torsten Hoefler, John Shalf, Mark Abraham, Mauro Bianco, Bradford L Chamberlain, Romain Cledat, H Carter Edwards, Hal Finkel, et al. 2017. Trends in data locality abstractions for HPC systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (2017), 3007–3020.

[53] S. Verdoolaege, M. Bruynooghe, G. Janssens, and P. Catthoor. 2003. Multi-dimensional incremental loop fusion for data locality. In *Proceedings of ASAP*.

[54] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. 1996. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *ASPLOS*.

[55] Nandita Vijaykumar, Eiman Ebrahimi, Kevin Hsieh, Phillip B Gibbons, and Onur Mutlu. 2018. The locality descriptor: A holistic cross-layer abstraction to express data locality in GPUs. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 829–842.

[56] Thomas Willhalm and Nicolae Popovici. 2008. Putting intel® threading building blocks to work. In *Proceedings of the 1st international workshop on Multicore software engineering*. 3–4.

[57] Michael E. Wolf and Monica S. Lam. 1991. A Data Locality Optimizing Algorithm. In *Proceedings of PLDI*.

[58] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of International Symposium on Computer Architecture (ISCA)*.

[59] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing Transformations on Hierarchically Structured Data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[60] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: Throughput-oriented Programmable Processing in Memory. In *Proceedings of HPDC*.

[61] Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. 2018. Minnow: Lightweight Offload Engines for Worklist Management and Worklist-Directed Prefetching. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

Received February 2021; revised April 2021; accepted April 2021