Bandwidth-Aware Loop Tiling for DMA-Supported Scratchpad Memory

Mingchuan Wu
SKL Computer Architecture, ICT, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
wumingchuan@ict.ac.cn

Ying Liu*
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
liuying2007@ict.ac.cn

Huimin Cui
SKL Computer Architecture, ICT, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
cuihm@ict.ac.cn

Qingfu Wei
SKL Computer Architecture, ICT, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
weiqingfu19s@ict.ac.cn

Quanfeng Li
SKL Computer Architecture, ICT, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
liquanfeng19f@ict.ac.cn

Limin Li
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
lilimin@ict.ac.cn

Fang Lv
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
flv@ict.ac.cn

Jingling Xue
School of Computer Science and Engineering, University of New South Wales
Sydney, NSW 2052, Australia
j.xue@unsw.edu.au

Xiaobing Feng
SKL Computer Architecture, ICT, CAS
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
fxb@ict.ac.cn

ABSTRACT

Scratchpad Memory (SPM) is widely used in emerging domain-specific architectures and accelerators for improving energy efficiency and time predictability. Typically, SPM-based architectures use DMA for fetching data from off-chip memory and global load instructions for loading fine-grained data directly into registers. For such architectures, neither capacity-only nor bandwidth-only loop tiling can efficiently use the bandwidth and SPM. This paper introduces a bandwidth-aware loop tiling approach that enables a tradeoff between SPM space utilization and bandwidth utilization to be made, by leveraging a runtime tiling framework and a cross-host-kernel IPA. Experimental results demonstrate that our approach can achieve the performance improvement of up to 4x, with a geometric average of 26%.

CCS CONCEPTS

• **Software and its engineering → Source code generation**.

KEYWORDS

Loop Tiling, DMA, Scratchpad Memory, Compiler

ACM Reference Format:
Mingchuan Wu, Ying Liu, Huimin Cui, Qingfu Wei, Quanfeng Li, Limin Li, Fang Lv, Jingling Xue, and Xiaobing Feng. 2020. Bandwidth-Aware Loop Tiling for DMA-Supported Scratchpad Memory. In *Proceedings of the 2020 International Conference on Parallel Architectures and Compilation Techniques (PACT '20), October 3–7, 2020, Virtual Event, GA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3410463.3414637

1 INTRODUCTION

Nowadays, Scratchpad Memory (SPM) is widely used as the on-chip memory for a number of domain-specific architectures [37, 54] and accelerators [9, 17, 20, 48]. Unlike caches, SPM, which is explicitly managed by software, is mapped into an address range that is different from the main memory. SPM is a popular alternative of cache due to its two advantages, higher memory efficiency and better time predictability [4]. For example, Sunway TaihuLight, Diannao and CELL all use SPM as the on-chip memory.

Moreover, emerging SPM-based architectures typically provide a DMA mechanism for fetching data to SPM [17, 20, 48] with a limited bandwidth, together with some global SPM-bypassing load instructions that can directly manipulate some individual elements

*To whom correspondence should be addressed

PACT '20, October 3–7, 2020, Virtual Event, GA, USA

ACM ISBN 978-1-4503-8075-1/20/10…$15.00
https://doi.org/10.1145/3410463.3414637

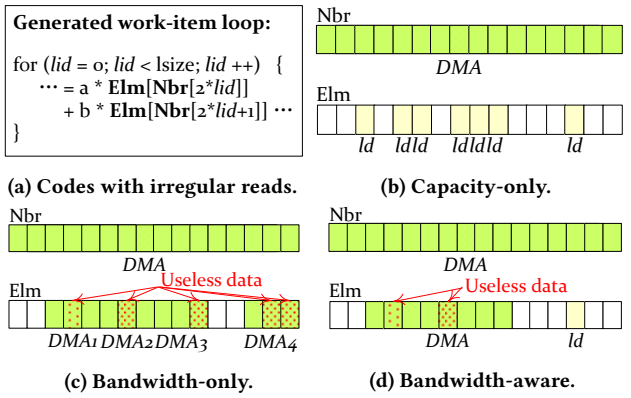


Figure 1: An example (simplified from SPECACCEL/*cfd*) for comparing the capacity-only, bandwidth-only and bandwidth-aware loop tiling.

in registers [20]. Unfortunately, such global load instructions usually waste significantly the available bandwidth, due to the long access latency caused by the pipeline stalls in the processor [64]. As a representative SPM-based accelerator, the core processor of the Sunway TaihuLight, SW26010, provides these two mechanisms, which often exhibit significantly diverse bandwidth utilization rates, ranging from 0.3GB/s to the peak 28GB/s.

Loop tiling, which is a significant optimization in compilers, is a loop transformation for exploiting the spatial and temporal locality of data accesses in loop nests. On cache-based architectures, the tile size for a loop nest is typically selected according to the cache capacity by computing its working set in order to keep the frequently used data at a given level of cache targeted. A great deal of research has been done on the tile size selection for cache-based architectures [12, 30, 39, 43, 67, 69, 74].

For SPM-based architectures, researchers have focused on data allocation and data fetching, by placing most frequently used data objects in SPM, only if they can be stored entirely in SPM [10, 47]. As a result, some frequently used arrays cannot be allocated in SPM if they are larger than SPM. In the case of data allocation, a representative work on data tiling, proposed by Li et al [33, 34], uses an ILP solver to select the best tiling schemes for all loop nests in the program as a whole. In the case of data fetching, Bhatotia et al [6] proposed a bounded method to fetch all indirect irregular accesses using DMA, by using a list of bounding boxes that contain all the needed elements. However, this method leverages a *bandwidth-only* policy and mandatorily uses DMA operations for data fetching, even if a large amount of useless data would end up being fetched, sometimes wasting a significant amount of SPM. By far, there is no analytical model to guide loop tiling by considering simultaneously both the bandwidth and capacity.

We have observed that for the DMA-supported SPM-architecture, neither capacity-only nor bandwidth-only approaches can efficiently use the bandwidth and SPM. We use the example in Figure 1 for illustration. In the example, one simplified loop nest of OpenCL code is selected from SPECACCEL/*cfd*, as shown in (a), containing two array accesses (*Nbr* and *Elm*). When we apply capacity-only tiling (as shown in (b)), *Nbr* will be fetched to SPM using a DMA operation, while the irregular *Elm* will be fetched to registers using global load instructions. This scheme will cause the overall bandwidth utilization to be very low, i.e., 4.6GB/s. On the other hand, when we apply bandwidth-only tiling in [6] (as shown in (c)), a large number of useless elements (shown by the dotted blocks) will be fetched due to the bounded box, achieving higher bandwidth utilization (28GB/s) but at the expense of significantly poorer utilization of SPM. In this paper, we propose a bandwidth-aware tiling approach that coordinately considers bandwidth and capacity, by fetching the green elements of *Elm* using a DMA operation (with 1 useless data shown by the dotted block) and reading the yellow block using load instructions, thus enabling a tradeoff to be made between the SPM capacity and bandwidth utilization (16.8GB/s).

Our loop tiling approach works by coordinately considering bandwidth and capacity as follows. First, we create a decision tree to select optimal data fetching operations for different data access patterns. Second, we build a dynamic loop tiling framework to compute the optimal tile size according to the selected data fetching operations, and adjust the tile size at runtime when necessary. Finally, a parameter-guided IPA is proposed to enhance the loop tiling, by seeking for opportunities of static tiling for irregular accesses and using re-computation for saving SPM capacity. We have implemented our approach in the Sunway TaihuLight's OpenCL compiler, SWCL, and demonstrate its performance benefits on the SW26010 chip, using the OpenCL programming model.

In summary, this paper makes the following contributions:

- *Bandwidth-aware loop tiling*. We develop a bandwidth-aware tile size selection model to simultaneously consider bandwidth and SPM capacity. We create a decision tree for modeling the bandwidth behavior, which is able to select different fetching operations for different segments of the same irregular-accessed array.
- *Runtime tiling framework*. We propose a dynamic loop tiling framework to first determine the optimal data fetching operations and tile size, and then generate proper data fetching statements, and finally adjust the tile size dynamically.
- *OpenCL-specific parameter-guided interprocedural analysis (IPA)*: We propose the IPA for analyzing the memory objects accessed in both the host and kernel codes, and find new optimization opportunities of static tiling for irregular accesses and using re-computation for saving SPM capacity.
- We implement the bandwidth-aware loop tiling approach SWCL, and evaluate it using the SPECACCEL [28] benchmark suite. Experimental results demonstrate that it can bring significant performance improvement, i.e., up to 4x, with a geometric average of 26%.

The rest of the paper is organized as follows. Section 2 introduces the background and motivation. Sections 3 and 4 describe the tile size selection model and the run-time framework of bandwidth-aware loop tiling, respectively. Section 5 describes OpenCL-specific parameter-guided IPA for further enhancing performance of bandwidth-aware loop tiling. Section 6 states our evaluation. Section 7 discusses the related work. Section 8 concludes.

2 MOTIVATION AND BACKGROUND

2.1 Sunway TaihuLight Overview

The Sunway TaihuLight supercomputer is powered by SW26010, and its architecture is shown in Figure 2. The processor has four core groups (CGs), each of which includes one management processing element (MPE), one computing processing element (CPE) cluster with 64 CPEs, one protocol processing unit (PPU), and one DDR3 memory controller (MC) [20] [16]. In this paper, we focus on the programming of one single CG.

Each MPE has a 32 KB L1 data cache and a 256 KB L2 instruction/data cache, and each CPE has its own 16 KB L1 instruction cache and a 64 KB SPM with the same speed as L1 cache. The data movement between main memory and the SPM can be performed through DMA with the theoretical bandwidth of 30.9GB/s for each CPE cluster, meanwhile, global load/store (*gld*/*gst*) instructions are provided for fine-grained data movements between main memory and registers with higher latencies, i.e., 177 and 278 cycles, respectively, achieving the bandwidth of 1.6GB/s.

2.2 Basic OpenCL Implementation on SW26010

As a unified heterogeneous parallel programming framework, OpenCL provides a platform-independent abstract *platform model*, enabling programmers to arrange computations and data references according to the OpenCL *execution model* and *memory model* [23]. These models are implemented in SWCL as follows.

The *platform model* consists of a host equipped with several OpenCL devices, with each device being divided into several compute units (CUs), which are further divided into several processing elements (PEs). On SW26010, we take the MPE as the host and the corresponding CPE clusters as the OpenCL device.

The *execution model* is defined in terms of two separate execution units, i.e., the kernel codes running on the OpenCL devices and the host codes on the host. When a kernel is launched for execution, an index space, i.e., NDRange is defined, in which each point is a work-item (kernel thread) running on one PE. The work-items are organized into work-groups with each work-group running on one CU, which is mapped to one CPE in SWCL. Thus the work-items are executed on the CPE serially, which is referred to as *serial execution mode* in [41]. Work-groups are statically assigned to CPEs using block distribution, by introducing an explicit loop nest, i.e., a *work-group loop* for each CPE, and work-group barriers are thus supported by loop fission, as in POCL [27], MOCL [72] and SNU-OCL [32].

The *memory model* defines two memory regions, the host memory and the device memory. Furthermore, the device memory consists of global/constant memory shared across all work-items, local memory shared in one work-group, and private memory used by one work-item only. On SW26010, both host memory and global/constant memory are mapped onto main memory, local memory onto SPM, and private memory to local variables that will be handled by the Sunway compiler backend.

2.3 An Motivation Example

In this section, we use the example of the loop nest in Figure 1 to demonstrate why capacity-only and bandwidth-only loop tiling approaches alone are not optimal for SPM-based architectures, which use DMA and load/store instructions to fetch data. The code accesses two buffers in an irregular way, i.e., *Nbr* for referring the subscripts and *Elm* for accessing the corresponding data.

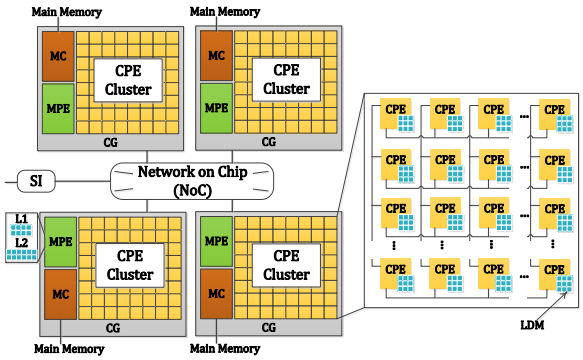


Figure 2: Architecture of SW26010

Figure 1(b) shows how to use the SPM when applying capacity-only tiling. In particular, the DMA operation will be leveraged to fetch the continuous *Nbr* accesses, and global load instructions will be used to fetch the individual irregular *Elm* accesses directly into registers. However, typically on SPM-based architectures, the global load instructions would cause very poor bandwidth utilization, i.e., 1.6GB/s on SW26010. Therefore, the overall bandwidth is only 4.6GB/s, leading to a significantly decreased performance even if the SPM is maximally utilized.

Figure 1(c) shows how to use the SPM when applying bandwidth-only tiling [6]. In particular, a list of fixed boxes are used to fetch all the accessed data into SPM. Here the box size is 3, thus fetching 7 useful elements and 5 useless elements using 4 DMA operations. In this case, the bandwidth utilization can be increased to 28GB/s, but $5 \times 4 = 20B$ of SPM capacity is wasted.

Considering the poor bandwidth utilization caused by capacity-only tiling and the SPM waste caused by bandwidth-only tiling, we can alternatively use a DMA operation (Figure 1(d)) for fetching the continuous green part of the *Elm* array, with 6 valid elements and 2 useless elements. Meanwhile, the left individual element (shown as yellow) is kept using global load instructions. Therefore, we can obtain the bandwidth of 16.8GB/s by wasting only 8B in SPM.

In summary, bandwidth-aware loop tiling is proposed to make a trade-off between bandwidth and SPM utilization. To this end, we need to address two research issues. First, how to select the optimal data fetching schemes for regular and irregular array accesses in the loop nest. Second, how to select the optimal tile size with the pre-determined data fetching operations.

3 BANDWIDTH-AWARE TILING MODEL

3.1 SW26010 Data Fetching Model

As stated in Section 2.1, the off-chip memory can be accessed in two ways, i.e., DMA and global load/store instructions (*gld*/*gst*). Furthermore, DMA can work in two modes, i.e., *continuous DMA*, which fetches one continuous data block for each CPE, and *strided DMA*, which fetches several blocks of the fixed size (*bsize*) with a

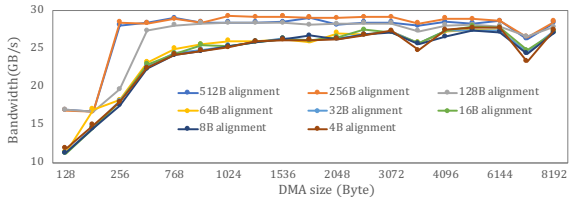


Figure 3: Measured DMA bandwidth for one CPE cluster.

fixed stride (*stepsize*) (Figure 4(a)). In either case, the amount of data that are fetched into SPM is denoted as *DMAsize*.

The above data fetching operations, i.e., continuous DMA, strided DMA, and *gld*/*gst*, exhibit different bandwidth utilization. In particular, continuous DMA can obtain a near-theoretical-peak bandwidth (28GB/s for each CPE cluster) when the data exceeds 1KB and is 256B-aligned (shown in Figure 3). Strided DMA will significantly reduce the bandwidth utilization when the block size (*bsize*) is small, e.g., only 0.3GB/s when *bsize* is 4B. Finally, *gld*/*gst* can achieve the bandwidth of about 1.6GB/s [64].

To measure the bandwidth utilization, we define the *effective bandwidth* (bw_{eff}) as its measured bandwidth bw multiplied by the ratio of useful data volume $volume_{useful}$ over all transferred data volume $volume_{all}$:

$$bw_{eff} = bw \times r_{eff}, \text{ where } r_{eff} = \frac{volume_{useful}}{volume_{all}} \quad (1)$$

3.1.1 Benchmarking DMA. For continuous DMA and *gld*/*gst*, the effective bandwidth can be determined. However, for strided DMA, it varies with *bsize*. We have thus designed a set of micro-benches to benchmark strided DMA, by using hardware DMA primitives. Our benchmarking is for one CG, in particular, we let its 64 CPEs launch the *same* DMA operations on data with *different* base addresses in order to simulate the OpenCL SPMD model, and we record the maximum execution time for computing the bandwidth.

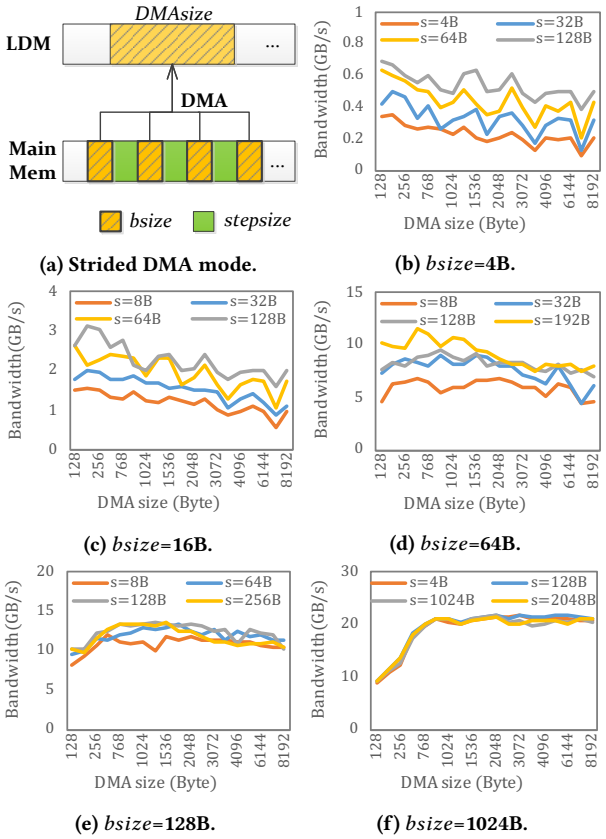
Figures 4(b)-(f) show the results of various *bsize* and *stepsize*(*s*). When *bsize* is larger than 1KB, it can be treated as a set of continuous DMAs, and a relatively high DMA bandwidth (22GB/s) can be achieved. However, a smaller *bsize* significantly reduces the effective bandwidth, e.g., only 0.3GB/s when $bsize = 4B$.

The bandwidth behavior for multiple DMA is modeled by synthesizing a set of n continuous DMA operations, with the i^{th} operation DMA_i having the size of dma_size_i ($i = 0, 1, ..n-1$). The measured bandwidth results are shown in Figure 5, plotted as the green line, which demonstrates that near-theoretical-peak (28GB/s) value can be obtained when *dma_size* exceeds 1KB.

Furthermore, for each DMA_i, we randomly split it into k multiple DMA operations ($k = 2..6$, respectively), i.e., DMA_i^j, where $i = 0..n-1$, $j = 1..k$, satisfying

$$\begin{cases} dma\_size_i = \sum_{j=1}^{k} dma\_size_i^j \\ dma\_size_i^j > 1KB \end{cases} \quad (2)$$

and compute the bandwidth using the execution time of all the DMA operations. The measured bandwidth for the k DMA operations is



(a) Strided DMA mode. **(b) *bsize*=4B.**

(c) *bsize*=16B. **(d) *bsize*=64B.**

(e) *bsize*=128B. **(f) *bsize*=1024B.**

Figure 4: DMA bandwidth of strided modes.

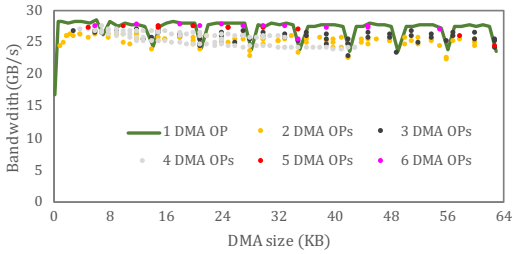


Figure 5: Bandwidth for multiple DMA operations.

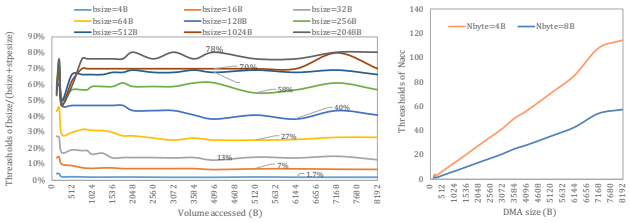
plotted as the dots in Figure 5, with five different colors for $k = 2..6$, respectively.

The results demonstrate that when multiple DMA operations are simultaneously executed, they can be aggregated to occupy the bandwidth, and near-peak bandwidth (28GB/s) can be obtained.

3.2 Decision Tree for Data Fetching

According to the bandwidth behavior model, we create a decision tree for selecting the optimal data moving operation to insert into the tiled loop nest, for maximizing the *effective bandwidth*.

- For a continuous memory access, we generate a continuous DMA operation for the accesses in the innermost loop nest. Furthermore, if the data volume accessed in the innermost loop nest is less than 1KB, the DMA operations from adjacent iterations of the innermost loop nest can be aggregated



(a) For strided accesses. (b) For irregular accesses.

Figure 6: Pre-calculated thresholds for tile size selection.

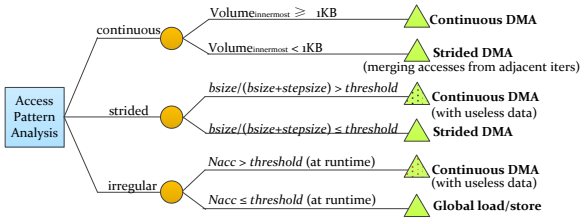


Figure 7: Decision tree for data fetching, in which decisions with dots apply to reading only.

together, into a strided DMA operation. Effective bandwidth can thus be increased since DMA startup costs are reduced.

- For a strided memory access, in some certain cases, we can fetch the useless stride data and use a continuous DMA operation for efficient bandwidth utilization, even if some SPM capacity will be wasted. In particular, for a strided access with the total volume *accvol*, we use the continuous DMA operation if $r_{eff} > threshold(bsize, accvol)$, where $r_{eff} = bsize/(bsize + stepsize)$ and *threshold* is computed using Figure 6(a), otherwise we use the strided DMA operation. Here, *threshold* is selected using the constant value marked for each *bsize*.
- For an irregular memory access reading N_{acc} locations with each location accessing N_{byte} data, we use a continuous DMA operation when $N_{acc} > threshold(N_{byte}, DMAsize)$, where *DMAsize* is the range of locations of all N_{acc} accesses times N_{byte}, and *threshold* is computed using Figure 6(b). Thus the benefit obtained due to more efficient bandwidth utilization would outweigh the cost of fetching useless data. Otherwise, we would use the *gld* instruction.

Figure 7 shows the final decision tree, in which the decision for continuous and strided accesses can be determined at compile-time, while for irregular accesses, some conditional branches are inserted into the generated tiled codes for determining the data fetching operations at runtime, as discussed in Section 4.

3.3 Tile Size Selection Model

After the optimal data fetching operations have been determined, the optimal tile size can be selected according to the SPM capacity. In particular, we compute the working set size *ws* for the tiled loop nest, by accumulating the *DMAsize* for all data moving operations, and *ws* can be represented as a function of the tile size *ts*. Therefore,

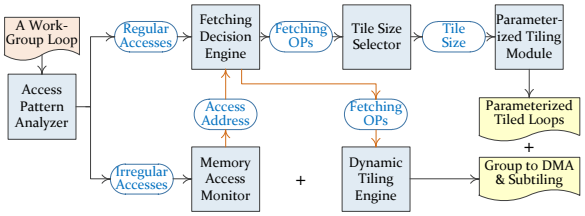


Figure 8: Compilation framework of bandwidth-aware loop tiling, in which a dynamic part is generated to tackle irregular accesses at runtime (by lines marked brown).

we select the optimal tile size using

$$opt\_ts = max\{ts | ws(ts) \leq C\} \quad (3)$$

where C is the SPM capacity.

When computing the working set size under a given tile size ts, i.e., $ws(ts)$, we traverse each memory object (*__global*) access. For a continuous or strided access, its *DMAsize* can be directly aggregated into $ws(ts)$. However, for an irregular access, e.g., A[B[i]], the *DMAsize* for B can be statically determined as $ts \times sizeof(B[i])$, while the *DMAsize* for A cannot be statically determined. For simplicity, we assume it as $ts \times sizeof(A[i])$ statically, and re-tile the loop nest for A at runtime when necessary, as discussed in Section 4.

3.4 Generalization to New Platforms

Bandwidth-aware loop tiling can apply to a platform with a DMA-supported SPM, since such a platform requires both capacity and bandwidth to be considered simultaneously. When applying the above three steps (Section 3.1 – 3.3), we only need to re-design the set of micro-benchmarks and model the bandwidth behavior (Section 3.1). The other two steps can be reused,as the decision tree (Section 3.2) and the tile size selection model (Section 3.3) are automatically created. Furthermore, the final code will also be automatically generated as described in Section 4.

4 BANDWIDTH-AWARE LOOP TILING FRAMEWORK

4.1 Framework Overview

The framework of bandwidth-aware loop tiling includes six components, as shown in Figure 8. Briefly, each *work-group loop* is analyzed by the *Access Pattern Analyzer* to divide array accesses into regular and irregular accesses. In particular, group references without dependencies are treated as accesses to different arrays on SPM, and those with dependencies are excluded from SPM. For regular accesses, they are fed into the *Fetching Decision Engine* to select an optimal data fetching operation statically, and the *Tile Size Selector* determines the optimal tile size, and the *Parameterized Tiling Module* generates the parameterized tiled codes following [52]. For irregular accesses, the *Memory Access Monitor* instruments the loop nest to collect the accessed addresses at runtime, and explores the *Fetching Decision Engine* to determine the optimal fetching operation on-the-fly, and finally the *Dynamic Tiling Engine* generates corresponding DMA operations and reduces the tile size when necessary.

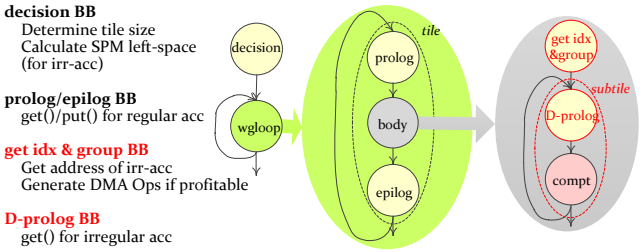


Figure 9: CFG for a *work-group loop* after tiling, with BBs inserted marked light yellow, including a static part (lines in black) and a dynamic part (lines in red).

4.2 Generated Tiled Code

Figure 9 shows a skeleton of the generated tiled codes, including a static part generated by the parameterized tiling module and a dynamic part generated by the memory access monitor and the dynamic tiling engine. Conceptually, the dynamic part works in a way similar as the classical inspector/executor method [44, 60].

For a regular access, first, it generates codes for computing the tile size using equation 3, shown as the *decision BB*. Second, it generates the parameterized tiled loop nest following [52], shown as the *body BB*. Finally, it generates the DMA operations selected according to the decision tree, shown as the *prolog/epilogBBs*.

For an irregular access A[B[i]], when computing the working set, we regard $ts \times sizeof(A[i])$ and $ts \times sizeof(B[i])$ as the working set sizes for A and B, respectively, and thus statically reserve the corresponding space on SPM for them. However, if the accesses of A are collected by the *get idx & group BB* and fed to the decision tree, which determines to use DMA operations for data moving, some extra useless data will be fetched, occupying extra SPM space thus the $ts \times sizeof(A[i])$ space is not enough. In this case, ts would be too large to keep the working set of one tile on SPM. Therefore, we need to decrease the tile size and re-tile the loop nests of one tile processing. We call this process as *subtiling*.

When applying subtiling, the new tile size is selected by recursively reducing it by half until the capacity constraint is satisfied. Then the *body BB* is re-tiled to generate the subtiled codes (for A). The subtile codes includes dynamically inserted DMA operations (*D-prolog BB*) and the computation codes (*compt BB*).

In particular, the instructions for DMA and subtiling are generated statically at compile time. These instructions are encapsulated in an if-statement and inserted into kernel codes, representing different code paths in the same CFG. At run time, when the if-condition is satisfied, these instructions will be executed.

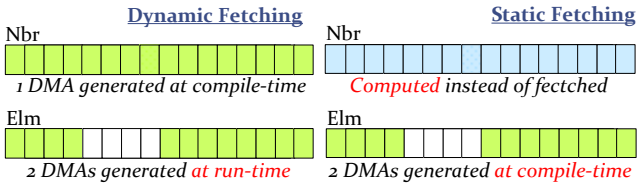
5 TILING WITH CROSS-HOST-KERNEL IPA

As described in Section 3, irregular accesses (*Elm* in Figure 1) can significantly benefit from being grouped into DMA operations. However, a necessary prerequisite is to analyze the memory addresses of these irregular accesses. Typically, we determine the addresses at runtime, obtaining the sub-tiling discussed in Section 4. If we can analyze the values of the subscripts of irregular accesses, we can determine the tile size statically to eliminate the

```
main (...) {
    cl_mem  d_nbr;
    int* h_nbr[2*N];
    for (i=0; i<N; i++) {
      h_nbr[2*N] = i/2;
      h_nbr[2*N+1] = i + N;
    }
    clWriteBuffer(d_nbr, h_nbr, 2*N*sizeof(int));
    clSetKernelArg(foo, 1, &d_nbr);
    clEnqueueNDRangeKernel(foo,...);
}
```

```
__kernel foo (__global float* Elm, __global int* Nbr, ...) {
    lid = get_local_id(0);
    ··· = a * Elm[Nbr[2*lid]] + b * Elm[Nbr[2*lid+1]] ···
}
```

(a) An example (abstracted from SPECACCEL/lavamd) that values of *Nbr* can be analyzed from host codes.



(b) Without IPA. **(c) With IPA.**

Figure 10: Cross-host-kernel IPA can help to analyze the irregular access patterns

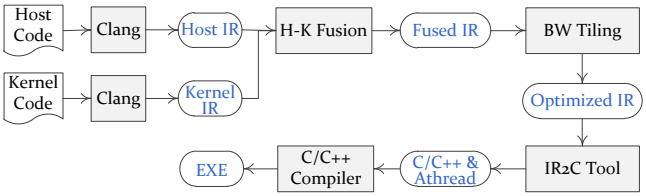


Figure 11: The Compilation flow of SWCL.

cost of sub-tiling. However, as these values are typically computed in the host codes, a a cross-host-kernel analysis is performed.

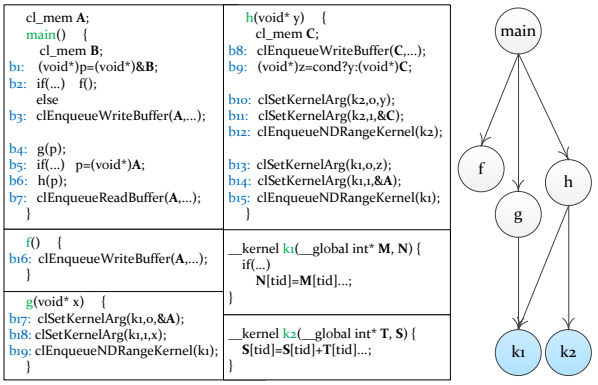
Figure 10 shows an example, with (a) listing the codes. In particular, (b) shows the result of bandwidth-aware tiling without any cross-host-kernel analysis, which first fetches *Nbr* into SPM to get the values, determines which elements of *Elm* will be used, and finally performs sub-tiling to determine the tile size. All these steps are performed at runtime. However, (c) shows the results with our cross-host-kernel analysis, where the values of *Nbr* are analyzed statically. Thus, the workset of *Elm* can also be determined statically, without introducing any runtime cost any more.

For the cross-host-kernel analysis, we propose a parameter-guided interprocedural analysis algorithm.

5.1 Parameter-guided Interprocedural Analysis

SWCL creates fused IRs from the host IRs and the kernel IRs, on which DMA-bandwidth-aware loop tiling is performed, as shown in Figure 11. After that, the optimized IRs are fed into the llvm-cbe tool [1] to generate optimized C/C++ codes with Athread library APIs, which are then compiled by the native C/C++ compiler (sw5cc) to produce executables.

When analyzing the fused IRs, our key insight is that we can focus only on those memory objects that have the type of *cl_mem*



(a) Host and kernel source codes. **(b) Call graph.**

Figure 12: An example for parameter-guided IPA.

and would be passed as actual parameters for launching some kernel. According to the OpenCL semantics, only these objects can be accessed by both host and kernel codes. We call such objects as *parameter-obj candidates*, and pointers that may point to those objects as *parameter-obj candidate pointers*. For this purpose, SWCL performs a *parameter-guided interprocedural analysis (IPA)* to analyze parameter-obj candidates and corresponding pointers.

Our parameter-guided IPA is designed based on the classical IPA [2], while makes two adjustments according to the OpenCL semantics. First, the analysis is confined to only parameter-obj candidates (and pointers). Second, the analysis leverages the semantic difference between host and kernel codes, e.g., the pointers in kernel codes are restricted according to [23], all function calls inside kernels will be inlined except built-in functions, and the host codes can only modify parameter-obj candidates (and pointers) and pass parameters to kernels via certain OpenCL APIs.

SWCL performs the parameter-guided IPA in three steps. (1) Generating fused IRs. The kernel compiler sends the generated kernel IRs to the host compiler via inter-process communication. Then the host compiler merges the received kernel IRs into the host IRs, to create fused IRs (without inlining) and the corresponding call graph. (2) Computing transfer functions (i.e., summaries). The host compiler performs a bottom-up traversing to the call graph and calculates the transfer function for summarizing the effect of each node, according to the corresponding node type. (3) Context-sensitive analysis. The host compiler performs a top-down context-sensitive analysis on the call graph, to propagate the caller information and get the global analysis results.

5.1.1 Generating Fused IRs. The host compiler and kernel compiler work coordinately to generate fused IRs.

The kernel compiler explicitly generates two-level loop iterations for traversing the work groups (the *work-group loop*) and the work items inside one work group (the *nested work-item loop*), as shown in Figure 13, and sends the obtained IRs to the host compiler via share-memory.

The host compiler merges the received kernel IRs into the host IRs, and generates the fused IRs. In particular, each instruction that has a host function call or kernel launching is expanded into a new

```
1 void example_kernel (...) {
2   CPE_begin = CPE_id * ceil(wg_num/64);
3   CPE_end = min(CPE_begin + ceil(wg_num/64), wg_num);
4   for (__group_id = CPE_begin; __group_id < CPE_end; __group_id++)  //work-group loop
5     for (__local_id_z = 0; __local_id_z < local_size[2]; __local_id_z++)  //nested work-item loop
6       for(__local_id_y = 0; __local_id_y < local_size[1]; __local_id_y++)
7         for(__local_id_x = 0; __local_id_x < local_size[0]; __local_id_x++)
8           ···; } //work-item body
```

Figure 13: The *work-group loop* and the *nested work-item loop* executed on each CPE.

Table 1: Summary of each function for codes in Figure 12(a).

n	$workset_n$	$live_n$.{MOD,REF,KILL}	$location_n$
$k1$	{M,N}	{01, 10, 00}	$\emptyset$
$k2$	{T,S}	{01, 11, 01}	$\emptyset$
f	{A}	{1, 0, 1}	{$b16$}
g	{A,x}	{01, 10, 00}	{$\emptyset$, {$b19$}}
h	{A,y}	{10, 01, 00}	{{$b15$}, $\emptyset$}
$main$	{A}	{1, 1, 1}	{$b3$, $b16$, $b15$}

BB, which is referred to as a *call site BB*. Meanwhile, the call graph is created, as shown in Figure 12.

5.1.2 Computing Summaries. The summary of a host function or kernel n, denoted as $summary_n$, indicates which parameter-obj candidates (and pointers) are modified/referenced/killed in it, which contains three issues:

- $workset_n$, represents the set of parameter-obj candidates (and pointers) that are externally visible when calling n.
- $live_n$, is defined as three bit vectors MOD,REF and KILL, with each bit representing one parameter-obj (or pointer) in $workset_n$. $\texttt{MOD}_i$, $\texttt{REF}_i$ and $\texttt{KILL}_i$ are set to 1 when $d_i \in workset_n$ is possibly modified, possibly used, and definitely killed by calling n, respectively.
- $location_n$, tracks the definition points for the parameter-obj candidates (and pointers), which is a set $\{A_i | i = 0, 1, ... workset_n.size() - 1\}$, with each A_i being a set of BBs that may include the definitions of $d_i \in workset_n$ by calling n.

If n is a kernel (which is a leaf node in the call graph), the set of its parameter-obj candidates $workset_n$ is defined as its formal parameters declared with *__global*, and $location_n$ is set to $\emptyset$. Table 1 shows the summaries for Figure 12(a) ($k1$ and $k2$ are kernels).

If n is a host function, things are a bit more complicated, as its $workset_n$ can consist of three types of objects:

- typeA: An object with the type of *cl_mem*, which is either a global variable or one of n's formal parameters / return values.
- typeB: A pointer that can be resolved in n to determine that it points to an object of typeA.
- typeC: A pointer that cannot not be resolved in n.

Algorithm 1 computes the summary for a host function. First, typeA and typeC are added into $workset_n$ (line 3). Second, we call the function of $GlobalParObj$ to summarize the external visible effects of $workset_n$ (line 4), which traverses all BBs seeking for function call instructions that define or use $\forall d_i \in workset_n$, by extracting the actual parameters $actual(n \rightarrow k)$ and applying parameter binding from $live_k$.MOD/REF to them (line 15). Thus the actual

parameters belonging to $workset_n$ are considered to have external visible definitions/uses (line 16). Meanwhile, if some OpenCL APIs are called, the analysis leverages the corresponding semantics (lines 19-27). Third, a typical iterative dataflow analysis is performed to compute $live_n$, and for $\forall d_i \in workset$, a set of BBs whose definition reaches out of the function are recorded in $loc_n(i)$ (line 5). Finally, $location_n$ is set by updating all *call host BBs* in loc_n (line 6).

Let's take the host function h in Table 1 for example, with global *cl_mem* A and formal parameter y added into $workset_h$. In the call to $k2$ at $b12$, $actual(h \rightarrow k2) = \{y, C\}$ and $live_{k2}$.MOD =01, but $C \notin workset_h$, hence $def(b12) = 00$ ($use(b12) = 01$, $kill(b12) = 00$). Similarly, for the call to $k1$ at $b15$, $def(b15) = 10$ ($use(b15) = 10$, $kill(b15) = 00$). Data-flow results at exit of h lead to $live_h$={10,01,00}, and $location_h = \{\{b15\}, \emptyset\}$.

5.1.3 Context-Sensitive Analysis. After the bottom-up summary computation (Section 5.1.2), SWCL performs a forward data-flow analysis by traversing the call graph top-down to propagate contexts from the callsites to the callees. In particular, the top-down phase analyzes all the objects that may be used as actual parameters in n for calling other functions, i.e., $parobj_n$, including $workset_n$ and some local memory objects, e.g., C in function h in Figure 12(a).

Algorithm 1 Compute host summaries

```
1:  Ws(n)/Live(n)/Loc(n): Map(n,workset_n/live_n/location_n)
2:  procedure SumHost(IR n)
3:      Ws(n) ← {cl_mem/void* in n's formals/ret-val or global}
4:      GlobalParObj(n, &def, &use)
5:      Live(n) ← DataFlowAnalysis(n, def, use, &loc_n)
6:      Loc(n) ← TrackCall(loc_n, Ws(n)) /* If a def point in
7:      loc_n is a call host BB, update it with callee's info */
8:  end procedure
9:  procedure GlobalParObj(IR n, Map &def, &use)
10:     IntraPointToAnalysis(n)
11:     for all CallsiteBB bb with callee k in n do
12:         /* actual(n → k): actual args when n calls k */
13:         /* Let [set A ↦ set B] yield a set C with B.size(),
14:            and C_i = B_i if B_i ∈ A, C_i = 0 otherwise */
15:         mod_act ← Live(k).MOD & actual(n → k)
16:         def(bb).get([mod_act ↦ Ws(n)]) = 1
17:         /* use(bb): similarly by Live(k).USE */
18:     end for
19:     for all BB bb calls clEnqueueWriteBuffer() do
20:         d ← call_arg(1) /* d is written */
21:         if d ∈ Ws(n) /* typeA and typeC */ then
22:             def(bb).get(d) = 1
23:         else if d.pointto() ∩ Ws(n) ≠ ∅ /* typeB */ then
24:             def(bb).get(d.pointto()) = 1
25:         end if
26:     end for
27:     /* use(bb): similarly by clEnqueueReadBuffer() */
28: end procedure
```

For a call site $cs{:}n \rightarrow k$, its context includes a bit vector ctx_{cs} and a set of definition points $ctxloc_{cs}$, denoting whether and where $actual(n \rightarrow k)$ are defined. During the top-down traversal, Algorithm 2 is used to compute contexts of all call sites in function n. First, contexts from all n's callers are unified as reaching definitions at n's entry (lines 3-8), and then data-flow analysis on $parobj_n$ is performed by processing summaries of n's callees and identifying *clEnqueueWriteBuffer()* (lines 9-16). After that, the context for $cs{:}n \rightarrow k$ is set to definitions of $actual(n \rightarrow k)$ reaching cs (lines 17-28). In particular, unresolved pointers in k (typeB) are processed (lines 20-22), e.g., if $p.pointto()$ is *cl_mem* {A,B} at cs, the reaching definitions of A and B are inserted into the context for $cs{:}n \rightarrow k$. Table 2 shows contexts of each call site in Figure 12(a).

Table 2: Context of each call site for codes in Figure 12(a).

$cs : n \rightarrow k$	$parobj_n$	ctx_{cs}	$ctxloc_{cs}$
$b2 : main \rightarrow f$	{A,B}	0	$\emptyset$
$b4 : main \rightarrow g$	{A,B}	10	$\{\{b16, b3\}, \emptyset\}$
$b6 : main \rightarrow h$	{A,B}	11	$\{\{b16, b3\}, \{b16, b3, b19\}\}$
$b19 : g \rightarrow k1$	{A,x}	10	$\{\{b16, b3\}, \emptyset\}\}$
$b12 : h \rightarrow k2$	{A,y,C}	11	$\{\{b16, b3, b19\}, \{b8\}\}\}$
$b15 : h \rightarrow k1$	{A,y,C}	11	$\{\{b16, b3, b19, b12\}, \{b16, b3\}\}\}$

Tiled work-item loop:

```
1  float* _spm_Elm1[ts/2];  /*ts: tile size*/
2  float* _spm_Elm2[ts];
3  for (t = 0; t < local_size[0]/ts + 1 ; t ++) {
4      _base_addr1 = t*ts/2;  /*Obtained by host codes analysis*/
5      _base_addr2 = t*ts + N;
6      DMA_get (_spm_Elm1, Elm[_base_addr1], (ts/2) * sizeof(float));
7      DMA_get (_spm_Elm2, Elm[_base_addr2], ts * sizeof(float));
8      for (lid = t * ts; lid < (t+1)*ts && lid < local_size[0]; lid++) {
9          idx1 = lid/2;  idx2=lid;      /*Inserted according to host codes*/
10         ··· = a * _spm_Elm1[idx1-_base_addr1] + b * _spm_Elm2[idx2-_base_addr2];
11     }
12 }
```

Figure 14: Tiling codes generated with cross-host-kernel analysis for codes in Figure 10(a).

5.2 Benefit to Loop Tiling

The parameter-guided IPA helps enhance the performance benefit of loop tiling, from two aspects.

First, it helps exploit static tiling opportunity for irregular accesses, thus avoiding the cost of runtime memory access monitoring and dynamic tiling. In Figure 10(a), the address range of B[A[i]] may be determined at compile-time, thus the DMA operations for fetching B are independent of the A[i]'s individual value. Figure 14 shows the generated tiled codes, where the values of $Nbr[2 \times lid]$ and $Nbr[2 \times lid + 1]$ are $lid/2$ and $N + lid$ respectively, hence Elm can be fetched using two DMA operations (lines 6-7).

Second, it helps select some candidates for re-computing before they are used, without occupying SPM space, potentially increasing the tile size and decreasing DMA operations. In Figure 14, *_base_addr1* and *_base_addr2* can be re-computed immediately before they are used, thus Nbr in Figure 10(a) does not need to be fetched to SPM any further (line 9).

We traverse the IR to seek for the above two optimization opportunities. An the irregular access A[B[i]] in kernel k can be optimized if all the following conditions are satisfied:

- B is not modified by k, i.e., $summary_k$.MOD.get(B)=0.
- $ctxloc_{cs}.get(B) = \{bb\}$ for all cs launching k.
- bb is a host BB and defines B with values of a host array hB using *clEnqueueWriteBuffer*.
- hB is calculated element-wise in a loop nest l using scalars, and without loop-carried dependencies.

Algorithm 2 Context-sensitive Analysis

```
1:  Ctx/CtxLoc:Map(cs,ctx_cs/ctxloc_cs)
2:  procedure CtxSensitiveAnalysis(IR n)
3:      for all CallSiteBB cs calls n do
4:          ctx_caller ← ctx_caller | Ctx(cs)
5:          loc_caller ← loc_caller ∪ CtxLoc(cs)
6:      end for
7:      D_in(n.entry() ← (ctx_caller, 0...0) /* D_in: Def reach in */
8:      LOC(n.entry()).get(Ws(n)) ← ctxloc_caller
9:      Parobj(n) ← Ws(n) ∪ {all local cl_mem A in n}
10:     AnalyzeParObj(n, &gen, &kill) /* Obtain gen/kill on
11:     Parobj(n) for each BB by summaries and OclAPIs */
12:     Iterative DFA with D_in(n.entry()) and gen/kill
13:     /* and obtain LOC:Map(BB, Set[defpoints]) in analysis */
14:     for all CallHostBB ch do
15:         LOC(ch) ← TrackCallLoc(LOC(ch), Parobj(n))
16:     end for /* Same as line 6 in algorithm 1 */
17:     for all CallSiteBB cs in n with callee k do
18:         for all ActualArg aa ∈ actual(n → k) do
19:             if aa is pointer and aa ∉ formals(n) then
20:                 pt ← aa.pointto() /* typeC for k */
21:                 Ctx(cs).get(aa) ← ∪_{p∈pt} D_in(cs).get(p)
22:                 CtxLoc(cs).get(aa) ← ∪_{p∈pt} LOC(cs).get(p)
23:             else
24:                 Ctx(cs).get(aa) ← D_in(cs).get(aa)
25:                 CtxLoc(cs).get(aa) ← LOC(cs).get(aa)
26:             end if
27:         end for
28:     end for
29: end procedure
```

The analysis in IPA is currently conservative to guarantee the safety of the optimizations enabled, and more optimization opportunities can be potentially found with more precise analysis.

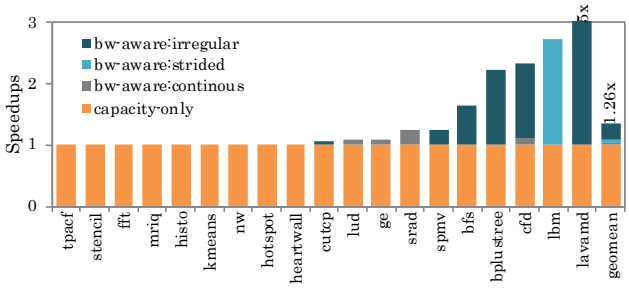
6 EVALUATION

We have implemented SWCL on LLVM-v9.0 using Clang-v3.9 as the front-end to produce LLVM IR and llvm-cbe [1] as the backend to generate C/C++ codes with Athread library APIs, which are compiled and linked by sw5cc (-O3).All our experiments are performed on one CG of SW26010 on Sunway TaihuLight, with the details discussed in Section 2.1.

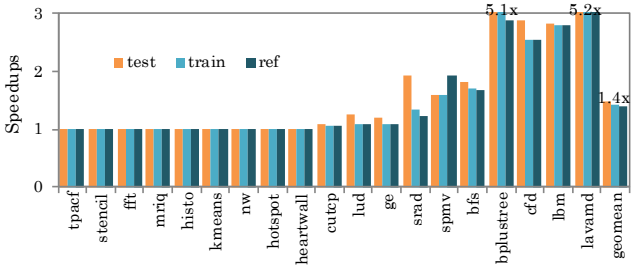
We use all the 19 OpenCL benchmarks in SPECACCEL-v1.2 [28], with each benchmark including one or multiple kernels. For each benchmark, we use its *test*, *train* and *ref* inputs, and present the execution times of the whole program and OpenCL kernels.

6.1 Overall Results over Capacity-only Tiling

Figure 15(a) shows the performance improvement of applying bandwidth-aware loop tiling, for the whole program (using the *ref* input) and in order of ascending speedups. Here, the baseline is capacity-only tiling with straightforward data fetching operations, i.e., continuous (strided) DMA for continuous (strided) accesses and *gld*/*gst* for irregular accesses. For the left nine benchmarks, SWCL failed to find data fetching optimization opportunities. For *lud*, *ge*, *srad* and *cfd*, SWCL optimized continuous memory accesses, achieving speedups of 1.08x, 1.09x, 1.23x and 1.09x, respectively. For *lbm*, SWCL optimized strided memory accesses, obtaining a speedup



(a) Speedups (whole program).



(b) Speedups (kernels).

Figure 15: Speedups of bandwidth-aware loop tiling over capacity-only loop tiling.

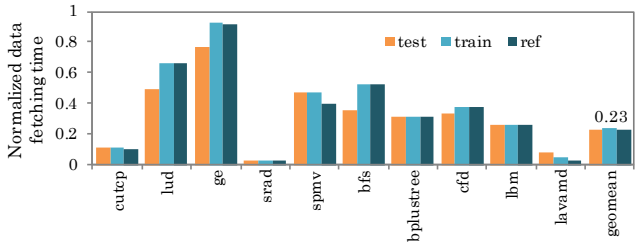


Figure 16: Total data fetching time of bandwidth-aware tiling, normalized to capacity-only tiling.

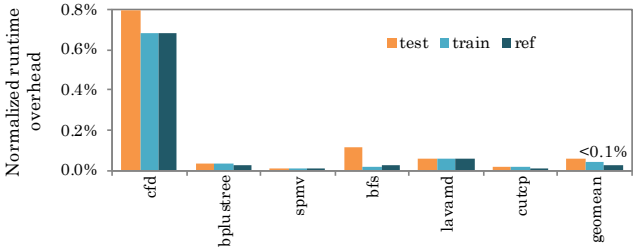
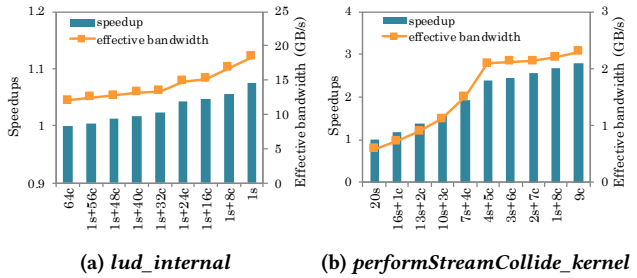


Figure 17: Runtime overhead for optimizing irregular memory accesses, normalized to kernel execution time.

of 2.7x. And for *spmv*, *bfs*, *bplustree*, *cfd* and *lavamd*, SWCL optimized irregular memory accesses, yielding significant performance improvements, ranging from 1.04x to 5x. Figure 15(b) shows the results for the kernels with *test*/*train*/*ref* inputs, which demonstrate that SWCL is effective across these different inputs.



(a) *lud_internal* (b) *performStreamCollide_kernel*

Figure 18: Speedups (the specified kernel) and effective bandwidth (for *lud* and *lbm*, using the *ref* input), where "*n*s+*m*c" denotes generating *n* strided-DMA and *m* continuous-DMA.

Furthermore, Figure 16 shows data fetching time of bandwidth-aware tiling, normalized to capacity-only tiling, for cases on which bandwidth-aware loop tiling finds optimization opportunities. Data fetching time is significantly reduced compared with capacity-only tiling, even if total data volume transferred may increase.

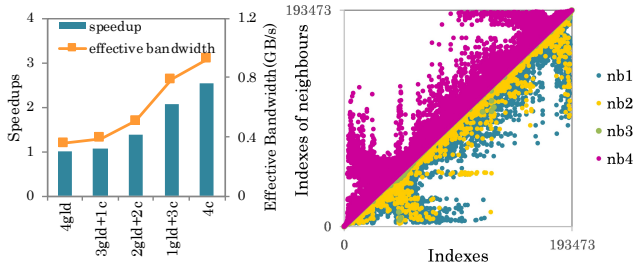
Figure 17 shows the overall runtime overhead for the cases optimized for irregular memory accesses, normalized to kernel execution time. Runtime overhead, which can be divided into three parts of collecting accessed addresses, grouping *gld* instructions into DMA, and selecting a subtiling size, takes less than 0.1% of the kernel execution time one average, is thus negligible.

6.2 Case Studies

We use four benchmarks to study the four typical cases that can be optimized by our bandwidth-aware loop tiling.

lud - continuous access. It includes a continuous access to a small volume of data in *lud_internal*, fetching a block of 64×64 elements from a matrix (buffer *m*) in each work-group. Therefore, SWCL merges the 64 continuous DMA operations into one strided DMA operation, and increases the bandwidth from 12.1GB/s to 18.4GB/s, yielding an 8% performance improvement for the kernel. We have measured the effective bandwidth for the kernel and plotted the results in Figure 18(a), for merging the 8, 16, 24, 32, 40, 48, 56 and 64 operations, respectively. The performance increases as more operations are merged together.

lbm - strided access. It includes accesses with a small *bsize* in *performStreamCollide_kernel*, which loads 20 discrete elements from buffer *srcGrid* (denote as $srcGrid[i]$, $i = 0, 1, ..19$) in each work-item. Therefore, capacity-only loop tiling selects the tile size of 4 (i.e., 400 work-items) for the *work-group loop*, issuing 20 strided-DMA operations with *bsize*=4B, *stepsize*=76B and *DMAsize* = 1600B and effective bandwidth 0.6GB/s. However, SWCL selects the tile size of 50 for the *work-item loop*, and transforms the 20 strided DMA into 9 continuous DMA with the effective bandwidth of 2.3GB/s, obtaining the speedup (*performStreamCollide_kernel*) of 2.8x, with the *ref* input. We have measured the effective bandwidth of this kernel and plotted the results in Figure 18(b), for merging 4, 7, 10, 13, 16, 17, 18, 19 and 20 operations respectively. Again, the performance increases as more operations are merged together. Furthermore, the left part in Table 3 shows the ratios of useful data volume to useless data volume for buffer *srcGrid* when merging various numbers of strided DMA operations, with the *ref* input.



(a) Speedups and bandwidth (b) Indexes of four neighbours

Figure 19: Analysis on *cfd* (using the *ref* input), with (a) Speedups (kernel *compute_flux*) and effective bandwidth, where "*n*gld+*m*c" denotes generating *n* *gld* instructions and *m* continuous-DMA. (b) Its irregular accesses.

Table 3: Ratios of useless data volume to useful data volume, for buffer *srcGrid* from *lbm* and buffer *variables* from *cfd*.

buffer *srcGrid*	*ratio*	buffer *variable*	*ratio*
20*s*	0	4*gld*	0
16*s*+1*c*	0.81		
13*s*+2*c*	1.67	3*gld*+1*c*	0.43
10*s*+3*c*	2.53		
7*s*+4*c*	3.39	2*gld*+2*c*	0.24
4*s*+5*c*	4.25		
3*s*+6*c*	5.21	1*gld*+3*c*	0.16
2*s*+7*c*	6.15		
1*s*+8*c*	7.10	4*c*	0.33
9*c*	8.05		

Results show that performance can be improved even if large SPM spaces are wasted, due to the increase of effective bandwidth.

cfd - irregular access. It includes an irregular access in *compute_flux*, which reads the 4 neighbours for each element in buffer *variables*. The capacity-only loop tiling would tile the *work-group loop* with the tile size of 3 (i.e., 576 work-items), and uses *gld* to load data. In comparison, SWCL selects the tile size of 2 (i.e., 384 work-items), and transforms 67% of *gld* instructions into continuous DMA operations, increasing the effective bandwidth from 0.35GB/s to 0.92GB/s, obtaining a speedup (*compute_flux*) of 2.3x, as shown in Figure 19(a). Sub-tiling is not triggered in this benchmark (for the *ref* input) because the 4 neighbors are closely located, as shown in Figure 19(b). Furthermore, the right part in Table 3 shows the ratios of useful data volume to useless data volume for buffer *variables* when transforming accesses to 1, 2, 3 and 4 neighbours from *gld* to DMA respectively, with the *ref* input, which indicate a relatively low amount of useless data.

lavamd - irregular access benefit from cross-host-kernel IPA. It includes a number of irregular accesses in *kernel_gpu_opencl*, all dependent on elements of buffer *d_box_gpu*, i.e., access to buffer *d_rv_gpu* and *d_qv_gpu* are determined by the field *nei* of elements from *d_box_gpu*, and accesses to buffer *d_rv_gpu* is also determined by the field *offset* of elements from *d_box_gpu*. The capacity-only loop tiling would tile the *work-group loop* with the tile size of 400 to just let buffer *d_box_gpu* stored in SPM. In bandwidth-aware loop

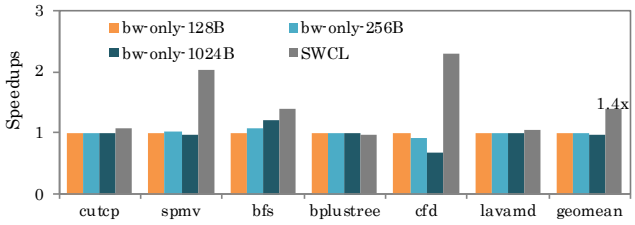


Figure 20: Speedups (kernels) of bandwidth-aware tiling over bandwidth-only tiling [6].

tiling, the tile size is reduced to one work-group to allow all the four buffers stored in SPM. Furthermore, buffer *d_box_gpu* is not modified in this kernel, and the two fields (i.e., *nei* and *offset*) are computed in host element-wise with scalars. SWCL analyses host codes and finds out that field *offset* grows linearly with *work_group_id*, and values of field *nei* denote 26 neighbouring elements with a maximum stride of 4 in x/y/z dimensions. Based on this analysis, SWCL succeeds to fetch data from buffers *d_rv_gpu* and *d_qv_gpu*, without depending on values of buffer *d_box_gpu* at run-time, and gains a speedup (whole program) of 1.09x compared with tiling without IPA, achieving a total speedup (whole program) of 5.2x over capacity-only loop tiling.

6.3 Comparison with Bandwidth-only Tiling

The work that is the most closely related to ours is [6], which uses a bounded method for irregular accesses, by dividing address ranges of irregular accesses into a number of sub-ranges with a fixed size *S*. We have implemented its algorithm on top of our tiling method, denoted as bw-only-*S*, where S = 128B, 512B and 1024B. Figure 20 shows the speedups (kernels) of SWCL over bw-only, for six benchmarks that contain irregular accesses, with the *ref* input and SWCL outperforms bw-only by 38%, 38%, and 43% for S = 128B, 512B and 1024B. In particular, significant speedups can be obtained for *spmv*, *bfs* and *cfd*, because they access small data fragments (e.g., 4B) irregularly, and bw-only would cause significant SPM waste.

7 RELATED WORK

Loop tiling policies. There has been a lot of work on applying loop tiling to improve parallelism and locality. To improve parallelism, researchers focus on concurrent execution of tiles [31, 65, 73], increasing parallelisms [11, 22, 25, 46], improving load balance [53] [59], and avoiding thread divergence [21]. In particular, to explore the parallelism of irregular applications, sparse tiling methods [29, 45, 56–58] are researched. To improve locality, researchers focus on increasing data reuse for various memory hierarchy [30] [66] [68], such as cache hierarchy on multicore architectures [12, 39, 43, 69, 74], registers and shared-memory on GPUs [13, 14, 26, 50, 51], by reducing the volume of data transfer between different levels in memory hierarchy. SWCL also performs loop tiling for increasing data reuse, but differs from them by balancing bandwidth and capacity. There are also some works on using model-driven empirical search methods [15, 63, 70] for tile size selection, as well as some machine learning techniques [42, 49, 71]. Also, works on parameterized loop tiling [5] [24] [25] [52] are used to facilitate iterative compilation and auto-tuning.

Loop tiling for SPM. [33] [34] perform SPM allocation and loop tiling considering aliases. DMATiler [35] focuses on regular accesses only, leaving irregular accesses to software cache. [32] uses a gather method to handle irregular accesses, which does not allow fetching useless data. Some works [7] [40] [6] use a bounded method for irregular accesses, fetching continuous data according to the address ranges of irregular accesses. Specifically, when SPM capacity is exceeded, [7] uses load instructions instead of DMA, whereas it's not discussed in [40] [6]. Comparing with them, we create a precise bandwidth-aware model for both regular and irregular memory accesses, and enable to fetch some useless data for increasing bandwidth utilization, by making a tradeoff between bandwidth utilization and SPM wasting.

Thread-fusion optimizations. There are a number of efforts on optimizing multiple threads simultaneously (i.e., multiple work-items in the same work-group [55] [32] [27]) to exploit different levels of parallelism, by performing various loop transformations and vectorization. These works typically optimize the threads executing on the same compute unit, whereas SWCL can further fuse the threads executing on different devices, e.g., host CPUs and accelerating cores.

Optimizations on Sunway architecture. There're a few works exploiting architectural features on Sunway, e.g., heterogeneous computing cores, SIMD, register-level communication, SPM, and so on, which are either hand-tuned application-specific implementations [3, 8, 19, 61], or domain-specific frameworks [18, 36, 75]. Specially, [38, 62, 76] perform hand-tuned tiling for parallelism.

8 CONCLUSION

In this paper, we have proposed an approach of bandwidth-aware loop tiling for OpenCL programs, which enhances the traditional loop tiling by coordinately considering the bandwidth utilization and on-chip memory capacity. First, a decision tree is created to select optimal data fetching operations for different data access patterns, according to the predicted bandwidth utilization. After that, a dynamic loop tiling framework which leverages the selected data fetching operations is used to determine the optimal tile size, and generate the parameterized tiled code at runtime. Furthermore, a parameter-guided IPA is exploited to enhance the loop tiling, by seeking for opportunities of static tiling for irregular accesses and using re-computation for saving SPM capacity. We implement the approach in the Sunway TaihuLight's OpenCL compiler, i.e., SWCL, and test it using SPECACCEL. Results show that bandwidth-aware loop tiling is more effective than capacity-only and bandwidth-only loop tiling, on architectures with DMA-supported SPM.

ACKNOWLEDGMENTS

We thank all the reviewers for their valuable comments and suggestions. This work was supported in part by the National Key Research and Development Program of China (2017YFB0202002), the Strategic Priority Research Program of Chinese Academy of Sciences (Grant No. XDC05030101), the National Natural Science Foundation of China (61802368, 61521092, 61432016, 61432018, 61332009, 61702485 and 61872043), CCF-Tencent Open Research Fund, and Australian Research Council grants (DP170103956 and DP180104069).

REFERENCES

[1] 2014. LLVM-CBE. https://github.com/JuliaComputing/llvm-cbe

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. 2011. *Compilers, Principles, Techniques and Tools* (2 ed.).

[3] Y. Ao, C. Yang, X. Wang, W. Xue, H. Fu, F. Liu, L. Gan, P. Xu, and W. Ma. 2017. 26 PFLOPS Stencil Computations for Atmospheric Modeling on Sunway TaihuLight. In *Proceedings of the 31th IEEE International Parallel and Distributed Processing Symposium (IPDPS '17)*. IEEE, Florida USA.

[4] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. 2002. Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES '02)*. New York, NY, USA, 73–78.

[5] M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J.Ramanujam, and P. Sadayappan. 2010. Parameterized Tiling Revisited. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '10)*. ACM, New York, NY, USA, 200–209.

[6] P. K. Bhatotia, S. K. Aggarwal, and M. Chaudhuri. 2009. A Compilation Framework for Irregular Memory Accesses on the Cell Broadband Engine. In *Proceedings of the 15th International Symposium on High-Performance Computer Architecture (HPCA '09)*. IEEE, North Carolina, USA.

[7] G. Chen, O. Ozturk, M. T. Kandemir, and M. Karaköy. 2006. Dynamic Scratch-Pad Memory Management for Irregular Array Access Patterns. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '06)*. Munich, Germany.

[8] J. Chen, R. Tan, and Y. Zhang. 2017. Heterogeneous Parallel and Distributed Optimization of K-means Algorithm on Sunway Supercomputer. In *Proceedings of the 15th IEEE International Symposium on Parallel and Distributed Processing with Applications and the 16th IEEE International Conference on Ubiquitous Computing and Communications (ISPA '17)*. IEEE, Guangzhou, China.

[9] T. Chen, Z. Du, J. Wang, C. Wu, and Y. Chen. 2014. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, Salt Lake City, Utah, USA.

[10] D. Cho, I. Issenin, N. Dutt, J. W. Yoon, and Y. Paek. 2007. Software Controlled Memory Layout Reorganization for Irregular Array Access Patterns. In *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'07)*. Salzburg, Austria.

[11] M. Christen, O. Schenk, and H. Burkhart. 2011. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium (IPDPS '11)*. IEEE, New Orleans, Louisiana, USA, 676–687.

[12] S. Coleman and K. S. McKinley. 1995. Tile Size Selection Using Cache Organization and Data Layout. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95)*. ACM, New York, NY, USA, 279–290.

[13] H. Cui, L. Wang, Y J. Xue, Yang, and X. Feng. 2011. Automatic Library Generation for BLAS3 on GPUs. In *2011 IEEE International Parallel Distributed Processing Symposium*. IEEE, 255–265.

[14] H. Cui, J. Xue, L. Wang, Y. Yang, X. Feng, and D. Fan. 2011. Extendable Pattern-oriented Optimization Directives. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE, Chamonix, France, 107–118.

[15] H. Cui, J. Xue, L. Wang, Y. Yang, X. Feng, and D. Fan. 2012. Extendable Pattern-oriented Optimization Directives. *ACM Transactions on Architecture and Code Optimization* 9, 3 (Oct. 2012).

[16] J. Dongarra. 2016. *Report on the sunway taihulight system.* Technical Report Tech Report UT-EECS-16-742. University of Tennessee.

[17] D. Fan, X. Ye, W. Li, and D. Wang. 2018. An Efficient Many-Core Processor for High-Throughput Applications in Datacenters. In *Proceedings of the 24th International Symposium on High-Performance Computer Architecture (HPCA '18)*.

[18] J. Fang, H. Fu, W. Zhao, B. Chen, W. Zheng, and G. Yang. 2017. swDNN: A Library for Accelerating Deep Learning Applications on Sunway TaihuLight. In *Proceedings of the 31th IEEE International Parallel and Distributed Processing Symposium (IPDPS '17)*. IEEE, Florida USA.

[19] H. Fu, J. Liao, and et al. 2016. Refactoring and Optimizing the Community Atmosphere Model (CAM) on the Sunway TaihuLight Supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE, Salt Lake City, Utah, USA.

[20] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, and et al. 2016. The Sunway TaihuLight supercomputer: System and applications. *Science China Information Sciences* 59 (2016), 1–16.

[21] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege. 2014. Hybrid Hexagonal/Classical Tiling for GPUs. In *Proceedings of the 12th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, Orlando, FL, USA.

[22] T. Grosser, A. Cohen, P. H. J. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege. 2013. Split tiling for GPUs: Automatic parallelization using trapezoidal tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU '13)*. ACM, 24–31.

[23] Khronos Group. 2018. OpenCL Overview. https://www.khronos.org/opencl/

[24] A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan. 2009. Parametric Multi-level Tiling of Imperfectly Nested Loops. In *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*. ACM, New York, NY, USA, 147–157.

[25] A. Hartono, M. M. Baskaran, J. Ramanujam, and P. Sadayappan. 2010. DynTile: Parametric tiled loop generation for parallel execution on multicore processorss. In *2010 IEEE International Symposium on Parallel Distributed Processing*.

[26] J. Holewinski, L. Pouchet, and P. Sadayappan. 2012. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12)*. ACM, Taiwan, China, 311–320.

[27] P. Jääskeläinen, C. S. Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg. 2015. Pocl: A Performance-Portable OpenCL Implementation. *International Journal of Parallel Programming*. 43, 5 (Oct. 2015), 752–785.

[28] G. Juckeland, W. C. Brantley, S. Chandrasekaran, and et al. 2014. SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance. In *Proceedings of 5th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS'14)*. Springer, New Orleans, LA, USA, 46–67.

[29] C. D. Krieger, M. M. Strout, C. Olschanowsky, A. Stone, S. Guzik, X. Gao, C. Bertolli, P. Kelly, G. Mudalige, B. Van Straalen, and S. Williams. 2013. Loop chaining: A programming abstraction for balancing locality and parallelism. In *Proceedings of the 18th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '13)*. Boston, Massachusetts, USA.

[30] M. S. Lam and M. Wolf. 1991. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI '91)*. ACM, New York, NY, USA, 30–44.

[31] R. Lazcano, D. Madroñal, E. Juarez, and P. Clauss. 2020. Runtime Multi-versioning and Specialization inside a Memoized Speculative Loop Optimizer. In *Proceedings of the 29th International Conference on Compiler Construction (CC '20)*. ACM, San Diego, CA, USA.

[32] J. Lee, J. Kim, S. Seo, S. Kim, and et al. 2010. An OpenCL Framework for Heterogeneous Multicores with Local Memory. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*. Vienna, Austria, 193–204.

[33] L. Li, L. Gao, and J. Xue. 2005. Memory Coloring: A Compiler Approach for Scratchpad Memory Management. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT '05)*.

[34] L. Li, H. Wu, H. Feng, and J. Xue. 2007. Towards Data Tiling for Whole Programs in Scratchpad Memory Allocation. In *Proceedings of the 12th Asia-Pacific Conference on Advances in Computer Systems Architecture (ACSAC '07)*. Miami Beach, Florida, USA.

[35] H. Lin, T. Liu, L. Renganarayana, H. Li, T. Chen, J. K. O'Brilen, and L. Shao. 2011. Automatic Loop Tiling for Direct Memory Access. In *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium (IPDPS '11)*. IEEE, New Orleans, Louisiana, USA.

[36] H. Lin, X. Tang, B. Yu, Y. Zhuo, W. Chen, J. Zhai, W. Yin, and W. Zheng. 2017. Scalable Graph Traversal on Sunway TaihuLight with Ten Million Cores. In *Proceedings of the 31th IEEE International Parallel and Distributed Processing Symposium (IPDPS '17)*. IEEE, Florida USA.

[37] Y. Lin, H. Lee, M. Woh, Y. Hare, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. 2007. SODA: A High-Performance DSP Architecture for Software-Defined Radio. *IEEE Micro* 27, 1 (2007), 114–123.

[38] C. Liu, B. Xie, X. Liu, W. Xue, H. Yang, and X. Liu. 2018. Towards Efficient SpMV on Sunway Many-core Architectures. In *Proceedings of the 32nd ACM International Conference on Supercomputing (ICS '18)*. ACM, Beijing, China.

[39] J. Liu, Y. Zhang, W. Ding, and M. T. Kandemir. 2011. On-chip cache hierarchy-aware tile scheduling form ulticore machines. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. ACM, Chamonix, France, 161–170.

[40] T. Liu, H. Lin, T. Chen, J. K. O'Brilen, and L. Shao. 2009. DBDB: optimizing DMATransfer for the cell be architecture. In *Proceedings of the 23rd international conference on Supercomputing (ISC '09)*. ACM, New York, NY, USA, 36–45.

[41] Y. Liu, L. Huang, M. Wu, H. Cui, F. Lv, X. Feng, and J. Xue. 2019. PPOpenCL: A Performance-Portable OpenCL Compiler with Host and Kernel Thread Code Fusion. In *Proceedings of the 28th International Conference on Compiler Construction (CC'19)*. ACM, Washington, DC, USA, 2–16.

[42] A. M. Malik. 2012. Optimal Tile Size Selection Problem Using Machine Learning. In *2012 11th International Conference on Machine Learning and Applications*, Vol. 2. 275–280.

[43] S. Mehta, R. Garg, N. Trivedi, and P. Yew. 2016. Leveraging Prefetching to Boost Performance of Tiled Codes. In *Proceedings of the 2016 International Conference on Supercomputing (ISC '16)*. ACM, New York, NY, USA.

[44] M. Mohammadi, T. Yuki, K. Cheshmi, E. Davis, M. Hall, M. Dehnavi, P. Nandy, C. Olschanowsky, A. Venkat, and M. Strout. 2019. Sparse Computation Data Dependence Simplification for Efficient Compiler-Generated Inspectors. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 594–609.

[45] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick. 2009. Minimizing communication in sparse matrix solvers. In *Proceedings of the 21th Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*. IEEE, Portland, Oregon, USA.

[46] R. T. Mullapudi, V. Vasista, and U. Bondhugula. 2015. Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, Istanbul, Turkey, 429–443.

[47] P. R. Panda, N. D. Dutt, and A. Nicolau. 1997. Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In *Proceedings of the 1997 European Conference on Design and Test (EDTC '97)*. IEEE Computer Society, USA, 7.

[48] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, and J. Keaty. 2005. The design and implementation of a first-generation CELL processor - a multi-core SoC. In *Proceedings of the 2005 International Conference on Integrated Circuit Design and Technology (ICICDT '05)*. IEEE, Austin, TX, USA.

[49] M. Rahman, L. Pouchet, and P. Sadayappan. 2010. Neural networks assisted tile size selection. In *5th International Workshop on Automatic Performance Tuning*.

[50] M. Ravishankar, J. Holewinski, and V. G. Forma. 2015. A DSL for image processing applications to target GPUs and multi-core CPUs. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs (GPGPU '15)*. ACM, 109–120.

[51] P. S. Rawat, C. Hong, M. Ravishankar, V. Grover, L. Pouchet, A. Rountev, and P. Sadayappan. 2016. Resource Conscious Reuse-Driven Tiling for GPUs. In *Proceedings of the 25th International Conference on Parallel Architectures and Compilation Techniques (PACT '16)*. Haifa, Israel.

[52] L. Renganarayanan, D. Kim, S. Rajopadhye, and M. M. Strout. 2007. Parameterized Tiled Loops for Free. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA.

[53] Y. Sato, T. Yuki, and T. Endo. 2019. An Autotuning Framework for Scalable Execution of Tiled Code via Iterative Polyhedral Compilation. *ACM Transactions on Architecture and Code Optimization* 15, 4 (Jan. 2019).

[54] S. Seo, R. G. Dreslinski, M. Woh, C. Chakrabarti, S. Mahlke, and T. Mudge. 2010. Diet SODA: A Power-Efficient Processor for Digital Camerasg. In *Proceedings of the 16th International Symposium on Low Power Electronics and Design (ISLPED '10)*. ACM, Austin, Texas, USA.

[55] P. Srivastava, M. Kotsifakou, and V. Adve. 2016. HPVM: A Portable Virtual Instruction Set for Heterogeneous Parallel Systems. https://arxiv.org/pdf/1611.00860.pdf

[56] M. M. Strout, L. Carter, , and J. Ferrante. 2003. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, New York, NY, USA.

[57] M. M. Strout, L. Carter, J. Ferrante, and B. Kreaseck. 2004. Sparse tiling for stationary iterative methods. *International Journal of High Performance Computing Applications* 18, 1 (2004), 95–114.

[58] M. M. Strout, F. Luporini, C. D. Krieger, and C. Bertolli. 2014. Generalizing Runtime Tiling with the Loop Chain Abstraction. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS '14)*. IEEE, New Orleans, Louisiana USA.

[59] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C. Luk, and C. E. Leiserson. 2011. The pochoir stencil compiler. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '11)*. ACM, New York, NY, USA, 117–128.

[60] A. Venkat, M. Hall, and M. Strout. 2015. Loop and Data Transformations for Sparse Matrix Code. *SIGPLAN Not.* 50, 6 (June 2015), 521–532.

[61] X. Wang, W. Liu, W. Xue, and L. Wu. 2018. swSpTRSV: a Fast Sparse Triangular Solve with Sparse Level Tile Layout on Sunway Architectures. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, Vösendorf/Wien, Austria.

[62] X. Wang, P. Xu, W. Xue, Y. Ao, C. Yang, H. Fu, L. Gan, G. Yang, and W. Zheng. 2018. A Fast Sparse Triangular Solver for Structured-grid Problems on Sunway Many-core Processor SW26010. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP '18)*. ACM, Eugene, OR, USA.

[63] C. Whaley, A. Petitet, and J. Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput* 27, 1 (2001), 3–35.

[64] Z. Xu, J. Lin, and S. Matsuoka. 2017. Benchmarking SW26010 Many-core Processor. In *Proceedings of the 31th IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW '17)*. IEEE, Florida USA.

[65] J. Xue. 1997. Communication-Minimal Tiling of Uniform Dependence Loops. *J. Parallel Distrib. Comput* 42, 1 (1997), 42–59.

[66] J. Xue. 1997. On Tiling as a Loop Transformation. *Parallel Processing Letters* 7, 4 (1997), 409–424.

[67] J. Xue. 2000. *Loop Tiling for Parallelism*. Kluwer International Series in Engineering and Computer Science, Vol. 575. Kluwer.

[68] J. Xue and C. Huang. 1998. Reuse-Driven Tiling for Improving Data Locality. *International Journal of Parallel Programming* 26, 6 (1998), 671–696.

[69] J. Xue, Q. Huang, and M. Guo. 2005. Enabling loop fusion and tiling for cache performance by fixing fusion-preventing data dependences. In *Proceedings of the 2005 International Conference on Parallel Processing (ICPP'05)*. 107–115.

[70] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. 2003. A Comparison of Empirical and Model-driven Optimization. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, New York, NY, USA, 63–76.

[71] T. Yuki, L. Renganarayanan, S. Rajopadhye, C. Anderson, A. E. Eichenberger, and K. O'Brien. 2010. Automatic Creation of Tile Size Selection Models. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '10)*. ACM, New York, NY, USA, 190–199.

[72] P. Zhang, J. Fang, C. Yang, T. Tang, C. Huang, and Z. Wang. 2018. MOCL: An Efficient OpenCL Implementation for the Matrix-2000 Architecture. In *Computing Frontiers Conference (CF'18)*. ACM, Ischia, Italy, 10.

[73] J. Zhao and A. Cohen. 2019. Flextended Tiles: A Flexible Extension of Overlapped Tiles for Polyhedral Compilation. *ACM Transactions on Architecture and Code Optimization* 16, 4 (2019).

[74] J. Zhao, H. Cui, Y. Zhang, J. Xue, and X. Feng. 2018. Revisiting Loop Tiling for Datacenters: Live and Let Live. In *Proceedings of the 32nd International Conference on Supercomputing (ICS '18)*. ACM, Beijing, China.

[75] M. Zhao, R. Liu, Y. Liu, K. Song, and D. Qian. 2016. Parallel Image Processing on the Sunway Many-core Processor. In *Proceedings of the 18th International Conference on High Performance Computing and Communications*. IEEE, Sydney, Australia.

[76] W. Zhao, H. Fu, J. Fang, W. Zheng, L. Gan, and G. Yang. 2018. Optimizing Convolutional Neural Networks on the Sunway TaihuLight Supercomputer. *ACM Transactions on Architecture and Code Optimization* 15, 1 (2018).