Taming the "Monster": Overcoming Program Optimization Challenges on SW26010 Through Precise Performance Modeling

Shizhen Xu*, Yuanchao Xu*, Wei Xue*‡§, Xipeng Shen†, Fang Zheng¶, Xiaomeng Huang‡§, Guangwen Yang*‡§

*Department of Computer Science, Tsinghua University

†Computer Science Department, North Carolina State University

‡ Ministry of Education Key Laboratory for Earth System Modeling, and Center for Earth System Science, Tsinghua University

§National Supercomputing Center in Wuxi, China

¶National Research Center of Parallel Computer Engineering and Technology, China

{xsz12, xyc15}@mails.tsinghua.edu.cn, xuewei@tsinghua.edu.cn, xshen5@ncsu.edu, zhengfangwww_2000@163.com, {hxm, ygw}@tsinghua.edu.cn

***Abstract*—This paper presents an effort for overcoming the complexities of program optimizations on SW26010, the heterogeneous many-core processor that powers Sunway TaihuLight, the world top one supercomputer. The solution centers around a *precise, static* performance model for modern many-core processor. Through a careful design that leverages the special properties of SW26010 and an effective treatment to massive parallelism, the model achieves a high accuracy, showing less than 5% average errors in estimating program execution performance. The precise performance model opens many opportunities for analyzing and guiding code optimizations. The paper demonstrates the usefulness by revealing a series of insights on the effects of some important code optimizations on SW26010. Moreover, it demonstrates that with such a precise performance model, it is feasible to replace empirical auto-tuning with static auto-tuning for optimizing regular loops on heterogeneous many-core systems. Such a replacement speeds up the tuning process by as much as a factor of 43 while keeping the tuning quality loss below 6%.**

I. Introduction

This paper presents our experience in building and exploiting precise architecture performance models on SW26010, the processor that makes Sunway TaihuLight the current top one supercomputer in the world, boasting a 125-PFLOPS double-precision peak performance, a 93-PFLOPS sustained LINPACK performance, and a 6.05 GFlops/W performance per Watt [1].

As a customized heterogeneous many-core processor, SW26010 is able to offer 3.06 TFLOPS double-precision floating-point performance and impressive power efficiency (10 Gflops/W). The design, on the other hand, complicates programming on SW26010. For instance, the main computing elements of SW26010 feature a cache-less design, and rely on programmers for data locality through careful, explicit data movements from the main memory to its on-chip scratchpad memory. It supports multiple methods for data loading and storing, including DMA, direct data transfers across registers, and normal "ld/st". Unlike GPU-like commonly seen many-core processors, all the cores on an SW26010 are independent computing units rather than forming some SIMD-like groups (e.g., warps on GPUs). For these features and flexibilities, a full capitalization of SW26010 could be tremendously beneficial, but at the same time, it is dauntingly difficult for common programmers to achieve.

A high-level programming model (SWACC) and a threading library (Athread) have been introduced to help. With them, programmers can write high-level code more easily. However, making the code efficient remains difficult. Built on Rose Compiler [2], the source-to-source SWACC compiler, plus the native compiler, contains a rich set of optimization techniques; however, the default optimizations by these compilers often leave a large room for improvements. Finding the best optimizations through empirical search is a time-consuming process given the complexities and flexibi1lities brought by SW26010.

Existing researches on Sunway [3]–[5] has mainly focused on accelerating a specific application by best utilizing the newly-introduced architecture features and the unprecedented parallelism. However, insights on the applications' performance and the interplay with underlying architecture are rarely revealed. For instance, optimizations such as data structure transformation (e.g. AoS to SoA) and double-buffer have been adopted, but why they give the amount of benefits and what upper-bound speedups they can ultimately provide remain not well understood.

The goal of this work is to build a precise performance model for SW26010 to provide insights and assistance to performance optimizations on SW26010. Specifically, it tries to provide a way to reveal what optimizations are beneficial, to analyze how much improvements the optimizations could yield, and to guide code tuning and optimizations on SW26010.

There have been many efforts in building up hardware performance models, with several recent ones focused on massive parallel architectures [6]–[8]. They give us insights, but do not directly fit our needs, for two reasons. First, most of the models (e.g., [6]) still require actual executions to collect some metrics (e.g., cache miss ratio, memory instruction counts, etc). Such needs limit the models in helping address the main drawback of empirical auto-tuning, being a time-consuming process. Second, they, especially those using only static code properties, are still subject to 10% to 23% performance estimation errors [8]. Such a level of accuracy limits their usefulness in guiding code optimizations.

In this work, we show that it is possible to create a static performance model for SW26010 with an average 95% accuracy. The success comes from the careful modeling of SW26010 on various data accesses and their interplay with computations and a simple clean treatment to the massive parallelism.

We apply the performance model in static tuning, which assesses the quality of an optimization based on the performance model rather than empirical performance measurement. We experiment with loop tiling and unrolling on regular loops, and compare the results with those from empirical auto-tuning. Our results show 1.8X–3.8X program performance improvements, which are within 6% performance loss compared to that by empirical auto-tuning. Meanwhile, the enabled static tuning reduces the tuning time by up to 43X, thanks to its avoidance of the many profiling runs required by the empirical tuning method.

Although performance models have been used in guiding code optimizations before, we have not seen such success of static auto-tuning purely on static performance models. The results validates the usefulness of the precise performance model, and at the same time, points out a promising direction for accelerating code optimizations on modern heterogeneous many-core systems.

Although this work concentrates on SW26010, some insights are potentially beneficial for other modern many-core optimizations, such as the methods of analyzing memory behavior of many-core architecture and memory/computation overlapping, the feasibility and promise of performance model-based static (auto-)tuning, and so on.

Overall, this work makes the following major contributions:

- It builds a static performance model for SW26010 that gives an average 95% accuracy.
- It shows the usefulness of the precise performance model in improving the understandings on the effects of program optimizations.
- It demonstrates that with such a precise performance model, it is feasible to replace empirical auto-tuning with static auto-tuning for optimizing regular loops on heterogeneous many-core systems.

II. The SW26010 Processor and Programming Model

The TaihuLight [1] is equipped with 40,960 SW26010 processors, taking the top place in the latest TOP500 lists since June 2016. This section presents the architecture of SW26010 processor and its current programming model.

A. The SW26010 Processor Architecture

Figure 1 shows the overview of SW26010 architecture. Each processor is composed of 4 *core-groups* (CGs). Each CG has 1 *management processing element* (MPE), 64 *computing processing elements* (CPEs) and a memory controller (MC). Each CG executes independently and four CGs are connected through a crossbar network on chip (NoC). The whole processor can provide a peak double-precision floating-point performance of 3.06 TFlops with 136 GB/s hardware memory bandwidth.

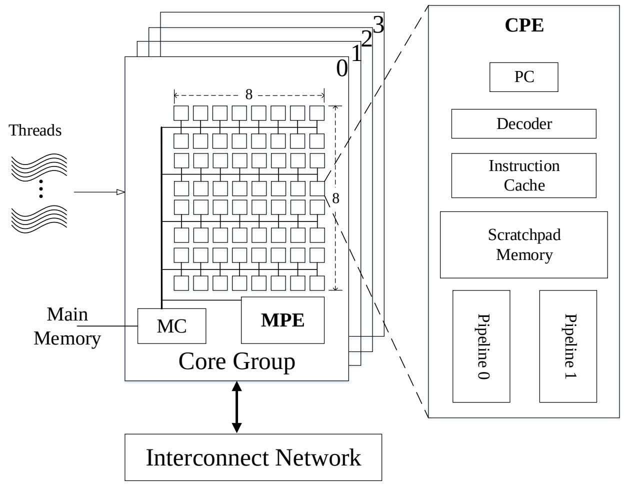


Fig. 1. An overview of SW26010 many-core processor architecture

The MPE is a complete out-of-order core that has a conventional 2-level cache system. In practice, it is used for communication and management. Each CPE contains a 64 KB *Scratchpad Memory* (SPM). The SPM is designed with an SRAM and configured as a user-controlled in-core memory. The address space of SPM is separated from that of DRAM, and programmers have to explicitly move data onto or out of the SPM. Each CPE also contains two execution pipelines: *pipeline 0* is responsible for floating point operations, DMA, and so on, while *pipeline 1* is for ”ld/st” operations, conditional branching, and others. Simple inter-pipeline out-of-order execution is supported by the CPEs.

The CPEs are organized as an 8 by 8 mesh. Every two rows of CPEs are connected with the memory controller through one data bus. SW26010 provides two different ways for CPEs to access main memory. The first is to move data between main memory and SPM through the dedicated DMA engine, which is referred as a ***DMA** request*. The programmer may specify a large (no larger than the SPM capacity) amount of data in one DMA request; the DMA engine transfers the requested data in

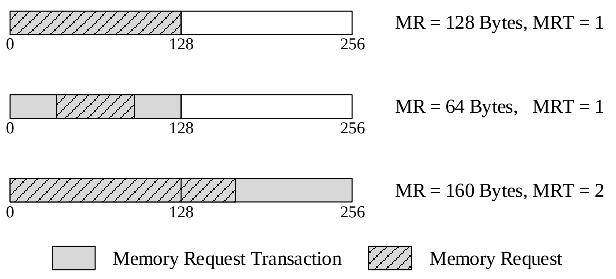
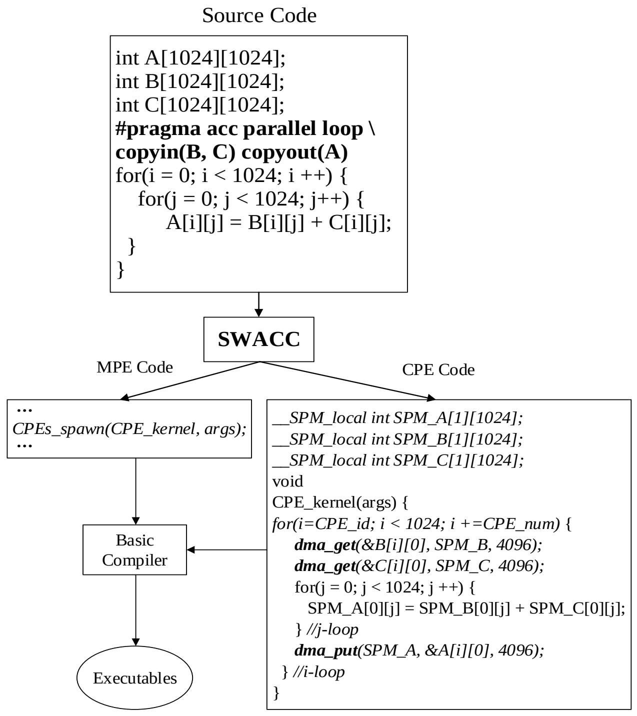


Fig. 2. Examples of the relations between Memory Requests (MR) and the needed Memory Request Transactions (MRT). All the three examples apply to DMA while only the middle one applies to Gload, because Gload can only request up to 32 bytes.



```
int A[1024][1024];
int B[1024][1024];
int C[1024][1024];
#pragma acc parallel loop \
copyin(B, C) copyout(A)
for(i = 0; i < 1024; i ++) {
    for(j = 0; j < 1024; j++) {
        A[i][j] = B[i][j] + C[i][j];
  }
}
```

```
...
CPEs_spawn(CPE_kernel, args);
...
```

```
__SPM_local int SPM_A[1][1024];
__SPM_local int SPM_B[1][1024];
__SPM_local int SPM_C[1][1024];
void
CPE_kernel(args) {
for(i=CPE_id; i < 1024; i +=CPE_num) {
    dma_get(&B[i][0], SPM_B, 4096);
    dma_get(&C[i][0], SPM_C, 4096);
    for(j = 0; j < 1024; j ++) {
       SPM_A[0][j] = SPM_B[0][j] + SPM_C[0][j];
    } //j-loop
    dma_put(SPM_A, &A[i][0], 4096);
  } //i-loop
}
```

Fig. 3. The workflow of the SWACC compiler

128-Byte blocks. The second is to move data between main memory and CPE registers with normal ”ld/st” instructions, which is referred as a ***Gload** request*. The Gload requests support up to 32-bytes data transfer within one request.

On cache-based architectures, the main memory is accessed in the granularity of cache lines. However, the CPEs of SW26010 are not equipped with cache system. They access the main memory in the unit of *DRAM transaction*. As a result, the occurred memory transactions actually reflect the effective DRAM throughput.

Noted that as data are transferred in blocks, some of the memory bandwidth has to be wasted if a block contains both useful and useless data. Figure 2 illustrates the relations between memory requests **(MR)** and the underlying memory request transactions **(MRT)** in several scenarios.

B. The Programming Model

During a program execution, one thread occupies one CPE and programmers can launch up to 64 threads per CG to maximize the usage of CPEs. *Athread* is a low-level interface for setting up threads on SW26010, akin to the POSIX Pthread library. The *Sunway OpenACC* (SWACC) is the high-level programming model for Sunway TaihuLight, focusing on data parallelism. The SWACC compiler is a source-to-source compiler. It supports the OpenACC2.0 specification, and automatically converts annotated programs into low-level Athread library calls (as shown in Figure 3).

The SWACC provides two key abstractions, the data decomposition and the SPM data placement. The data decomposition represents the number of data elements for each CPE, as well as the total number of CPEs in use. The SPM data placement dictates what specific data shall be put into and out of the SPM. There are three related intrinsic operations, *copyin*, *copyout*, and *copy*, which respectively indicate the data to be copied into SPM, to be copied out from SPM, or to be copied in both directions.

Taking Vector-Add (Figure 3) as an example. The source code indicates arrays A, B and C are to be copied into SPM. The SWACC compiler distributes the workload to the CPEs along dimension i (outer-most loop) and each thread performs all the computation throughout dimension j. Furthermore, the data composition is pre-computed and statically assigned to each CPE according to its core-id. In this example, the active CPE number is $64 = min(1024, CPEs\ per\ CG)$, and each CPE is responsible for a sub-region of $\frac{1024}{64} \times 1024$ and needs such an amount of data. In the meantime, SWACC introduces a critical ***tile*** intrinsic. Here, “tile” differs from loop tiling. It does not do loop tiling, but decides the data copy granularity. In this vector-add example, without an explicit indication of tile, each CPE computes along dimension i in a round-robin way, which means the copy granularity equals 1024 elements. However, if we apply *tile*$(j : 32)$ to the inner loop, the arrays are copied to SPM at a granularity of 32 elements. If we apply *tile*$(i : 32)$ to the outer loop, each CPE copies the array at a granularity of 32×1024 elements. In the latter case, the total number of CPEs that actively involve in the executions of the program (denoted as $\#active\_CPEs$) would equal $\frac{1024}{32} = 32$. It is because the tile intrinsic decides that each CPE has to process 32 consecutive data elements along the i dimension.

The workflow of SWACC compiler is also shown in Figure 3. It converts the SWACC programs to the MPE and the CPE source code, readable for programmers. The MPE source code controls the MPE-CPE interactions and task concurrency, and the CPE source code controls data transfer, the SPM allocation and the CPE calculation. For consecutive data copy, the SWACC generates one DMA call. As for stride data copy, it generates several DMA calls, with each corresponding to a consecutive chunk of data. Finally, with the help of the low-level compiler of SW26010, the converted source code is compiled into executables. From the CPE code in Figure 3, we can conclude that the programs for SW26010 architecture are usually composed of 3 parts, data copy to SPM, execution (both computation and Gload requests), and data copy to memory.

III. The Static Performance Model

This section describes the precise static performance model we have built for SW26010. The model, built on the SWACC programming model, is the key component for evaluating different program optimizations as well as enabling the static auto-tuning for SWACC compilers. For typical SW26010 applications, most computations are usually put into some CPE kernel functions, which are the focus of optimizations and hence the focus of the performance modelling.

The performance model predicts the execution time of application kernels running on CPEs of SW26010. Its design features a simple yet effective treatment to massive parallelism through two introduced concepts (*memory request parallelism* and *virtual grouping*), a careful modeling of the various data accesses and their overlapping with computations.

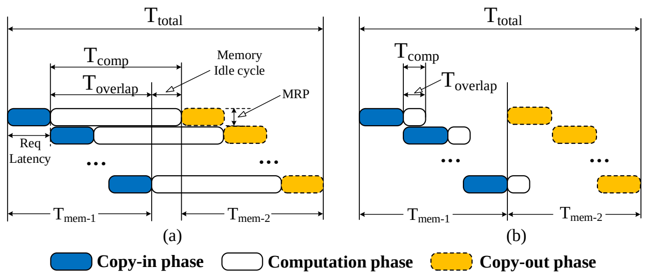


Fig. 4. A typical SWACC application execution process in the perspective of virtual grouping. Figure (a) refers to Scenario 1 (Section III. A) in which computation time is longer than memory access time, while figure (b) refers to Scenario 2. The height of each colored block represents MRP, which is the capacity of one virtual group, and the number of colored blocks represents the number of virtual groups.

A. Introduction to MRP and the Overlapping Model

The SW26010 processor is an SMP architecture, and unlike GPUs where threads form SIMD warps, all CPEs execute computation instructions independently, contending for the limited memory bandwidth. So a key part of an accurate performance model is to precisely model the memory contentions and the overlapping between memory accesses and computations.

A major challenge is the massive parallelism and non-deterministic execution orders of many independent CPEs. We address it by introducing the concepts of *Memory Request Parallelism* (MRP) and *virtual grouping*. MRP is defined as the number of memory requests that can be concurrently issued and served among the simultaneous memory requests; *virtual grouping* refers to the MRP CPEs that their threads can issue memory requests at the same time and the requests are served by the DRAM engine equally. After a virtual group of CPEs win the memory contention, they are able to continue the subsequent computations, and another virtual group of CPEs gets into the memory access.

MRP and virtual grouping significantly simplify the modeling complexities due to the massive non-deterministic parallelism of many independent CPEs. Our experiments (Section V) show that the simplification introduces little inaccuracy. With them, it becomes easier to analyze memory contentions among CPEs and models the overlapping effect between memory accesses and computations. The memory-computation overlaps are regarded to happen between the memory accesses by one virtual group and the computations by other virtual groups.

To accurately calculate the MRP, we must be aware of the memory bandwidth and the MRT (i.e., the number of memory request transactions as explained in Section II-A). Rather than calculating the requested data size, we calculate the number of requested memory transactions as it is a much more precise approximation for memory throughput.

As described in Section II-B, a typical execution process on SW26010 can be divided into three parts, copying data to the SPM (via DMA), doing the computations, and copying data from the SPM (via DMA). Take the process in Figure 4 as an example. Active CPEs are partitioned into $NG = \frac{\#active\_CPEs}{MRP}$ groups. The first group completes the memory access in a period of $Req\_Latency$ while the last group completes in a period of $Req\_Latency \times NG$. In our model, $Req\_Latency$ (the blue block in Figure 4) is a function of MRT and baseline latency (Equ 11) and the memory bandwidth should be considered in calculating NG (Equ 9). During the interval of $(NG-1) \times Req\_Latency$, the first group is able to issue the subsequent computation. Comparing the computation time T_{comp} and the interval $(NG-1) \times Req\_Latency$, we can categorize the execution process into 2 scenarios.

1) Scenario 1: $(NG-1) \times Req\_Latency < T_{comp}$: As shown in Figure 4(a), when the last virtual group completes copying data to SPM, the first group is still in the computation phase. The computation time is only partially overlapped with the memory copy-in time of $NG-1$ groups and there exist memory idle cycles. When the first group finishes its computation, the copy-out phase begins. In this case, the overlapping time $T_{overlap}$ can be denoted as $(NG-1) \times Req\_Latency$ from the viewpoint of CG.

2) Scenario 2: $(NG-1) \times Req\_Latency \geq T_{comp}$: As shown in Figure 4(b), when the last MRP group completes copying data to the SPM, the first group has already finished the computation. The computation is fully overlapped with memory access. When the last MRP group completes copying-in data, the first group begins copying-out data to main memory. In this case, there is no memory idle cycles. The overlapping time $T_{overlap}$ equals the computation time T_{comp}.

B. Breakdown of Total Execution Time

As a whole, the execution time T_{total} of a program can be divided into memory access time T_{mem} and computation time T_{comp}. The memory access time can be further divided into Gload request time T_g and DMA request time T_{DMA}. Combined with the overlapping effect, the predicted total execution time can be defined as follows:

$$T_{total} = T_{mem} + T_{comp} - T_{overlap} \tag{1}$$

$$T_{mem} = T_g + T_{DMA} \tag{2}$$

T_{comp} represents the time spent on issuing computation instructions by a single CPE. It is worth noting that as all CPEs execute the same kernel functions, they typically exhibit similar lengths of executions. Upon load imbalance, the longest execution time among the CPEs is used for T_{comp}.

C. Gload Request Time (T_g) and DMA Request Time (T_{DMA})

The Gload/DMA request time ($T_{g/DMA}$) is the sum of the latency (L) of each request, which is measured with the number of cycles. If all active CPEs access the main memory with no contention, this latency equals the baseline latency, which is a processor-related parameter (as in Table I). However, given the limited memory bandwidth, some serializations may occur for the concurrent memory accesses. $T_{g/DMA}$ is hence modelled as follows:

$$T_{g/DMA} = \sum_{r}^{reqs} \max(L_{base},\ L_{mem\_bw,r}) \tag{3}$$

$$L_{mem\_bw,g/DMA} = \frac{\#active\_CPEs \times MRT_{g/DMA}}{mem\_bw/Freq/Trans\_size} \tag{4}$$

$$MRT_{g/DMA} = \lceil \frac{Req\_size_{g/DMA}}{Trans\_size} \rceil \tag{5}$$

In Equation 3, L_{base} represents the baseline latency of memory instructions and $L_{mem\_bw,r}$ represents the serving duration if memory bandwidth is fully utilized. In Equtions 4 and 5, the $MRT_{g/DMA}$ denotes the MRT of one Gload/DMA request of a single CPE. Gload request is generally regarded as the smallest granularity of memory access and it is able to support up to 32 bytes. As a result, the Gload request size (Req_size_g) is much smaller than the memory transaction size and thus MRT_g is usually equal to 1. On the contrary, DMA request size (Req_size_{DMA}) is much bigger and thus MRT_{DMA} usually surpasses 1. Req_size_{DMA} can be easily calculated from SWACC *copy* intrinsics. Under SWACC programming model, we regard the copy of all arrays in one *copy* intrinsic as one request. It is because each CPE halts at the last DMA instruction and waits for the completion of all the prior DMA requests. This is the procedure enforced (through code generation) by the SWACC compiler. Underlying data layout has to be taken into special considerations for stride DMA transfers. In such a scenario, the DMA request leads to more transactions than actually requested. It is worth noting that, Gload requests are typically very costly, because only a small portion of the memory bandwidth is utilized.

D. The Computation Time, T_{comp}

T_{comp} represents the computation time of one CPE. T_{comp} is calculated through the analysis of the compute instructions retired and *Instruction Level Parallelism* (ILP). The ILP refers to the pipeline effect of compute instructions in Pipeline 0 (shown in Figure 1). In our model, the compute instructions contain the floating-point, fix-point instructions and SPM access instructions. Different types of compute instructions may have different latencies as Table I shows. The cache-less architectural design of SW26010 helps avoid most non-determinism in instruction latency, making accurate estimations of ILP possible. The computation time (T_{comp}) is modelled as follows

$$T_{comp} = \frac{\sum_{t}^{comp\_inst\_type} L_t \times \#t}{avg\_ILP} \tag{6}$$

In Equation 6, the latency of type t of compute instructions (L_t) might be different, as listed in Table I. Programmers are assumed to be aware of the computation domain size and thereby the upper bound of each loop in the SWACC kernels is able to be figured out in advance. The native compiler annotates elaborately on the assembly code, including the predicted issue cycle of each instruction, the instruction dependency and the code basic-blocks. The number of compute instructions retired ($\#t$) can be counted through an analysis of the assembly code and assembly annotations are currently checked by programmers. Instruction level parallelism (ILP) regards the pipelined compute instructions. For the consecutive floating point instructions with no dependency, they can be fully pipelined and the ILP can be as many as 8 [1], as in Table I. The average ILP (avg_ILP) can then be calculated by counting the predicted execution time of each basic block.

E. The Overlapping Time, $T_{overlap}$

The overlapping time ($T_{overlap}$) between memory accesses and computations is the key component for performance modelling. The formulas are as follows:

$$T_{overlap} = \min(T_{comp}, T_{DMA\_overlap} + T_{g\_overlap}) \tag{7}$$

$$T_{g/DMA\_overlap} = (1 - \frac{1}{NG_{g/DMA}}) \times (1 - \frac{1}{\#g/DMA\_reqs}) \times T_{g/DMA} \tag{8}$$

$$NG_{g/DMA} = \lceil \frac{active\_CPEs}{MRP_{g/DMA}} \rceil \tag{9}$$

$$MRP_{g/DMA} = \frac{L_{avg\_g/DMA} \times mem\_bw}{Freq \times Tran\_size \times avg\_MRT_{g/DMA}} \tag{10}$$

$$L_{avg\_g/DMA} = L_{base} + (avg\_MRT_{g/DMA} - 1) \times \Delta delay \tag{11}$$

$$avg\_MRT_{DMA} = \frac{\sum^{reqs} MRT_{DMA}}{\#DMA\_reqs} \tag{12}$$

In Equ 10 and 11, avg_MRT_g is 1, as discussed in Section III-C.

To better explain them, we reinterpret Figure 4 with the terms in equations. $\#DMA\_reqs$ equals 2 with the first representing the copy-in request and the second representing

[1]The div/sqrt instructions are only partially pipelined, for simplicity, we do not consider the pipeline effect of these two kinds of instructions

TABLE I
THE SUMMARY OF MODEL PARAMETERS

	Input Parameters	Definition	Source
1	$\#active\_CPEs$	# CPEs in use	◇Sec II-B
2	$\#g\_reqs$	# Gload requests	◇∗Sec III-C
3	$\#DMA\_reqs$	# DMA requests	◇Sec III-C
4	$\#c\_inst$	# compute instructions retired	∗Sec III-D
5	avg_ILP	avg # pipelined compute instructions	∗Sec III-D
6	mem_bw	memory bandwidth per Core Group	32 GB/s
7	$Freq$	SW26010 processor freq	1.45 GHz
8	$Trans\_size$	DRAM transaction size in bytes	256 bytes[2]
9	$\Delta delay$	extra delay by one transaction request	50 cycles
10	L_{base}	baseline latency of memory access	220 cycles
11	$L_{floating}$	floating point operation latency	9 cycles
12	L_{fixed}	fixed point operations latency	1 cycle
13	L_{SPM}	SPM access latency	3 cycles
14	$L_{div/sqrt}$	divide and sqrt operation latency	34 cycles

	Output Parameters	Definition	Source
1	T_{total}	Total execution time	Equ 1
2	T_{mem}	Memory access time	Equ 2
3	$T_{g/DMA}$	Gload requests time	Equ 3 - 5
4	T_{comp}	Computation time	Equ 6
5	$T_{overlap}$	Overlapping time	Equ 7 - 12
6	$\#MRT_{g/DMA}$	Gload/DMA request transactions	Equ 5
7	$\#MRP_{g/DMA}$	Gload/DMA request parallelism	Equ 10
8	$\#NG_{g/DMA}$	Number of Gload/DMA MRP groups	Equ 9

◇: source code analysis; ∗: assembly code analysis

the copy-out request. As in Equ 7, either the whole computation (Figure 4(b)) or a portion of the memory request time (Figure 4(a)) is overlapped. DMA requests are served concurrently with computation, except for first virtual group (the first term of Equ 8), and the last DMA request (the second term of Equ 8). The request latency (L_{avg_DMA}, the blue block of Figure 4) is the baseline memory access latency plus an extra delay as in Equ 11.

Further, MRP refers to the one virtual group size and $\#MRP$ requests can fully utilize memory bandwidth during the period of request latency. The reflection of them in Figure 4 is that one virtual group (MRP CPE threads) behaves in lock-step way. Finally, NG represents the number of virtual groups, and the calculation is as in Equ 9.

F. Limitations

Our performance model does not consider the usage of MPE. As most computations have been offloaded to the CPEs, MPE is primarily responsible for launching the kernels. The MPEs in SW26010 can achieve only 1/65 of the peak computation performance; the benefit of using MPE is hence negligible. In addition, the effect of the instruction-cache has not been taken into consideration yet.

The effect of executing branch instructions, which may lead to imbalanced workload, is not modelled in details. Combination with some lightweight profiling is a feasible way to complement the static model to address the complexity. Currently we pick the most time consuming branch for the computation cost analysis.

IV. ANALYSIS OF PROGRAM OPTIMIZATIONS ON SW26010

One practical usage of the performance model is directly analyzing the effects of some optimizations. We discuss it with three examples. We will show that the analysis based on the performance model leads to some conclusions contradicting prior optimization guidelines.

1) The Effects of DMA Request Granularity: Prior optimization guidelines [4] suggests that enlarging the DMA granularity and using the SPM as much as possible can obtain better performance. However, our model shows that decreasing the DMA granularity is actually better as long as $DMA\_req\_size \geq Trans\_size$. According to our overlapping calculation equation (Equ 8), when each DMA request gets smaller and hence the number of DMA requests increases, the total overlapping time increases, and thereby the total execution time decreases (Equation 1). The overall time saving brought by using smaller DMA request granularity is formulated as follows (without considering the change of T_g, NG):

$$\Delta T_{smaller\_DMA} = \left(\frac{1}{\#DMA\_1} - \frac{1}{\#DMA\_2}\right) \times T_{DMA} \qquad (13)$$

where, $\#DMA\_1$ is the numbers of DMA requests before reducing the DMA request granularity, and $\#DMA\_2$ represents the number after.

2) The Effects of Double-Buffer Optimizations: The double-buffer optimization leverages two data buffers, one for the current computation and the other to hold the data being transferred with DMA. It is a popular memory and computation overlapping strategy for many-core architectures (e.g., the asynchronous memory transfer in CUDA [9]).

Figure 5 illustrates the workflow of double-buffer optimization. The performance improvements actually come from issuing subsequent computation earlier, which means the $T_{overlap}$ is further enlarged. However, the maximum possible performance improvement is the non-overlapped DMA time of one virtual group, as shown in the left of Figure 5. In

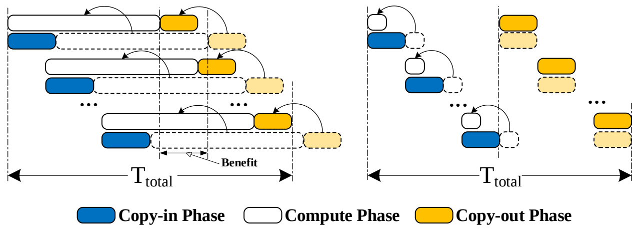


Fig. 5. The illustration of double-buffer optimization according to the analysis of the performance model. For the left part (Scenario 1 in Section III. A), the benefit is regarded as the copy-in duration of one virtual group. The right part (Scenario 2 in Section III. A) implies the double-buffer optimization does not yield performance improvement.

[2]While DMA requests for transferring data from main memory to CPEs are in 128-Byte unit (physical memory transaction size) , the memory page hit rates are often too low to fully use the memory bandwidth, hence the 256-Byte minimum transaction size is suggested by the architects and used in our model as shown in Table I.

the worst case, this optimization may bring no performance improvements, as the right part of Figure 5 illustrates.

The overall time savings by this optimization for SW26010 can be formulated as follows:

$$\Delta T_{db} = \min(\frac{1}{NG_{DMA}} \times T_{DMA}, T_{comp} - T_{overlap}), \qquad (14)$$

In a common scenario, the $\#active\_CPEs$ equals 64 while the DMA block is significantly larger than 256 bytes, we can derive from Equation 9 to 11 that NG_{DMA} = 16, , the benefits from double buffer is only 1/16 of the T_{DMA}. It means the acceleration of the whole program can be no better than 1/16. Section V validates the analysis results through experiments.

3) The Effects of #active_CPEs.: Although using more active CPEs increases computation performance, our performance model indicates that it does not necessarily lead to a higher overall performance.

Equation 5 shows that when $DMA\_req\_size < Trans\_size$, MRT_{DMA} always equals 1. Therefore, if the number of DMA requests increases, more redundant data would be transferred. This is important when determining the number of active CPEs. In many programs such as the WRF dynamics cases in Section V, more active CPEs could make DMA_req_size smaller than $Trans\_size$. In that case, reducing $\#active\_CPEs$ could help. Considering both the decrease of T_{DMA} (because of a decrease of $\#active\_CPEs$ in Equation 4) and the increase of T_{comp} (because of a increase of $\#comp\_inst$ in Equation 6), the overall time savings of using less active CPEs can be inferred as follows (without considering the change of T_g and $T_{overlap}$):

$$\Delta T_{less\_CPEs} = \Delta active\_CPEs \times max(0, T_{DMA} - T_{comp}) \qquad (15)$$

where, $\Delta active\_CPEs$ is the reduction fraction of $\#active\_CPEs$. This relation can be used to choose the appropriate number of active CPEs. From Equation 15, we can see that the benefit appears only when 1) T_{DMA} increases with more CPEs used (because of a waste of memory transactions); 2) $T_{DMA} > T_{comp}$ in the baseline. It should be noted that for problems with no waste of memory transactions on 64 CPEs, using all CPEs gives close-to-best performance. But in cases such as the WRF dynamics kernels in Section V, the best performance occurs when less than 64 CPEs are used, which is consistent with the analysis results by our model.

V. Experiments

The experiments consist of three major parts. The first part evaluates the accuracy of the performance model. We compare the estimated execution time and wall-clock execution time. The second part evaluates the benefits for analyzing the effects of program optimizations brought by the accurate performance model. The third part reports the observations of the static auto-tuning powered by the performance model.

A. Experiment Setup

Our experiments use the OpenMP kernels of the Rodinia benchmark suite [10]. The suite contains both regular and irregular programs, and covers a wide range of computations and memory access patterns. Here, *irregular* means 1) the memory access pattern is unpredictable so that conventional blocking techniques cannot be used, or 2) there could be imbalanced workload among CPEs. Based on the existing OpenMP implementation, we ported and tuned these benchmarks on SW26010 platform with SWACC and Athread, including the optimizations to maximize the usage of SPM on SW26010 to save memory access overhead. We also experiment on two real-world scientific kernels from WRF [11].

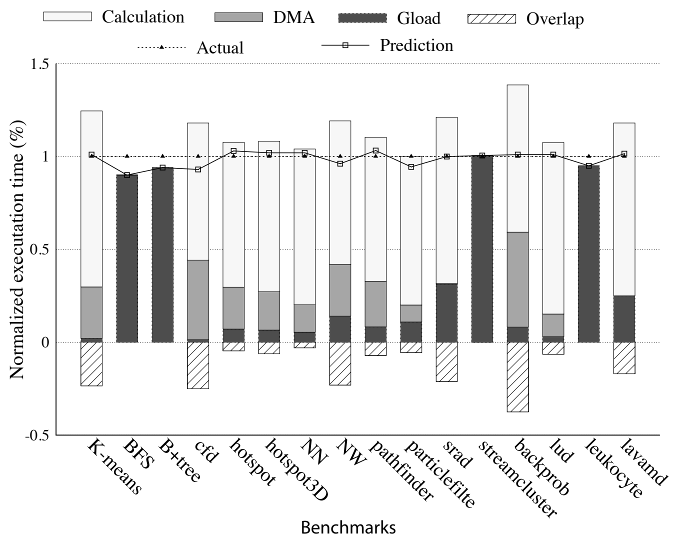


Fig. 6. Model prediction on Rodinia benchmark. We made prediction on the time breakdown of each benchmark. The positive stack values represent the predicted T_{comp}, T_{DMA}, T_{gload} from top to down, and the negative stack represents $T_{overlap}$. Actual and predicted T_{total} execution times (with Equation 1) are shown by lines. For simplicity, the actual execution times of all kernels are normalized to 1.

B. Evaluation of the Analytical Model

Figure 6 shows the breakdown of the predicted execution time of each benchmark, and the error between the predicted and the actual execution times. The times are all normalized by the actual execution times.

The average error is 5% and the errors vary across the benchmarks. The errors are marginal for regular benchmarks. The data accesses of K-Means, for example, are predictable, which allows a flexible task/data partition and the avoidance of Gload operations. The DMA granularity can also be adjusted to maximize the overlapping of the computation with the DMA data transfers. Such regularity leads to the near perfect prediction result.

The max error is 9.6% happening on BFS which is a typical irregular program. Similarly, memory accesses in `B+tree`, `leukocyte`, and `streamcluster` are difficult to leverage SPM. Conventional blocking techniques are invalid on such cases and thus we cannot move data onto SPM in advance. As a result, almost all the memory requests are Gload operations, which dominate the execution time. It is worth noting that the irregular features can also lead to the imbalanced workload among CPEs, which is not modelled in detail in this work. We take the longest execution path during analysis and the performance model assumes the amount of computation is able to be calculated in the beginning. The experimental results demonstrate that irregular computations on such a cache-less architecture suffer from the overhead of Gload (a waste of memory transactions) and need further optimizations to coalesce memory accesses. For some other kernels that are more amenable for DMA operations (e.g., `K-Means`, `cfd`, `backprob`), the DMA overlaps the computations substantially.

C. Enabling Analysis of Optimization Effects

1) DMA Request Granularity: As demonstrated in Section IV-1, DMA request granularity may affect the memory/computation overlapping and hence affect the execution time. Our experiments on the K-Means kernel consider two scenarios: *fixed input data size* (Figure 7(a)) and *different input data sizes* (Figure 7(b)).

In 7(a), we fix data elements processed by each CPE to 256. As the granularity of a request becomes smaller, the number of requests increases. According to Equation 8, the overlap is positively correlated with #DMA_reqs. Therefore, the increased number of requests causes an increase in overlap time and hence the overall performance. The whole kernel is accelerated by up to 20% when $\#data\ elements/DMA\ request$ is reduced from 256 to 32. An interesting discovery is that when the $\#data\ elements/DMA\ request$ decreases to less than 16, the Gload memory request increases sharply. It is because the assembly code generated by native compiler reveals there are additional Gload requests. The cause may be sophisticated but our model only calculates the number of global requests based on the native compiler and it can capture such cases. The total execution time increases according to Equation 1 and 2.

In 7(b), we fix the granularity to 256. As we increase the number of the data partition per CPE, the normalized execution time decreases. Similar to the reason of 7(a), it is because the number of DMA requests per CPE increases and hence the overlap time increases.

2) Double Buffering: On the N-body kernel, the double-buffer optimization shortens the exeution time from 1142us to 1100us, only yields a 3.7% (42us) improvement. Such a small effect has been qualitatively predicted by our analysis given in Section IV-2. Figure 8 provides the quantitative prediction from our performance model. The predicted benefit from double buffering matches well with the actual, with only a 3.3% error.

3) Proper #active_CPEs: We evaluate different $\#active\_CPEs$ on two kernels in a public weather forecasting application WRF [11] : one comes from the `WRF dynamics` and the other from `WRF physics`. They are memory-intensive and computation-intensive respectively. Figure 9 shows the predicted and actual execution time at a spectrum of $\#active\_CPEs$, while Figure 10 shows the breakdown of the actual execution time.

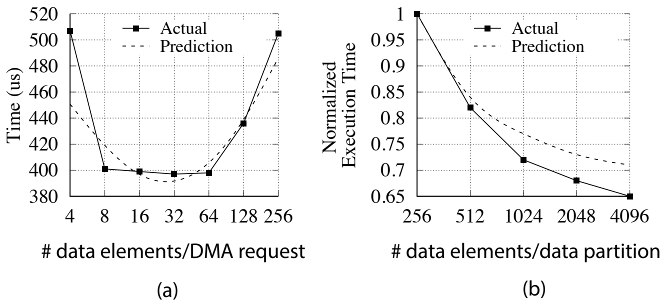


Fig. 7. The effects of varying DMA request granularity. (a) Actual and predicted execution time at different DMA request sizes. (b) Normalized actual and predicted execution time at different data partition sizes on a fixed DMA request size (256 elements). Data partition means the size of workload, and also infers the number of DMA requests.

`Predicted time(us) of optimized version:` T_{opt}=1098.6 (T_{comp}=975, T_{DMA}=695, T_{gload}=0, $T_{overlap}$=571.4, NG_{DMA}=8)
`Predicted benefits:` 1142us (original) - 1098.6us (opt) = 43.4us
`Actual benefits:` 1142us (original) - 1100us (opt) = 42us.

Fig. 8. Prediction of the benefits of double-buffering optimization

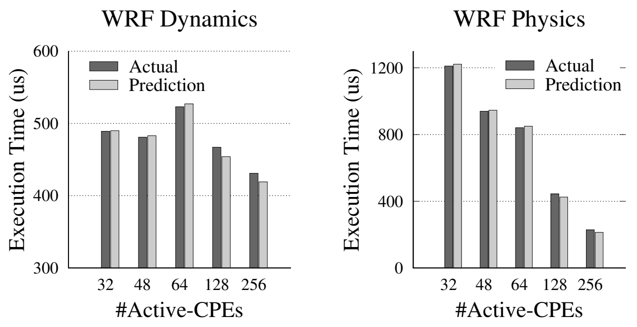


Fig. 9. The actual and predicted execution time with different $\#active\_CPEs$ for WRF Dynamics kernel (left part) and WRF Physics kernel (right part).

From Figure 9, we can see that for the memory intensive `dynamics` kernel, using 48 CPEs outperforms using 64 CPEs by about 10%. The time breakdown of `dynamics` in Figure 10 confirms the trade-off between the opposite effects of using more active CPEs on T_{comp} and on T_{DMA} as our performance model predicts in Section IV-3. As `physics` is computation-intensive, using more active CPEs reduces overall execution time. Our performance model gives accurate predictions of the execution time of both kernels as shown in Figure 9.

When $\#active\_CPEs$ surpasses 64, multiple CGs are used. On one hand, more DMA transaction are wasted if using more CPEs. On the other hand, the memory bandwidth and the computation performance increase. To scale programs to multiple CGs on SW26010 and enable sharing, data is allocated on *cross-section* memory, which is evenly (in a

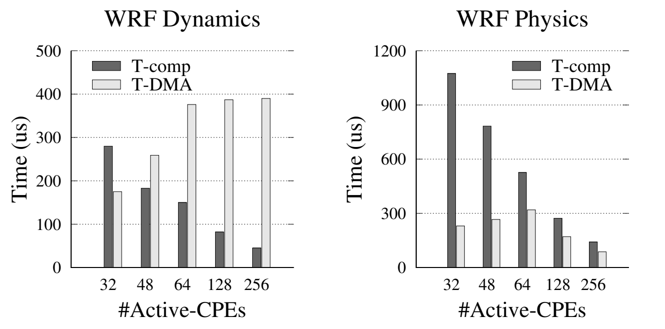


Fig. 10. Measured time breakdown for WRF Dynamics kernel (left part) and WRF Physics kernel (right part) when $\#active\_CPEs$ varies.

round-robin way) mapped to physical memory associated with each CG. Our tests show the memory bandwidth of cross-section is only slightly lower than the local memory. We believe the reasons are two-fold: 1) Because the bandwidth of cross-bar Network-on-Chip(NoC) is much larger than that of transfer engines within the CG, the bandwidth of cross-bar NoC is not the bottleneck. 2) For memory-bound kernels (e.g., the WRF dynamics kernels), they are typically bounded by memory bandwidth rather than latency for the massive parallelism of the system. Hence in our modelling, memory bandwidth (mem_bw) in Equation 4 and 10 increases linearly with the numbers of CGs used. The performance model appears to work for multiple CGs.

D. Enabled Static auto-tuning

To further validate our performance model and demonstrate the benefits and the opportunity of static tuning powered by the model, we set up an experimental static auto-tuner for selecting the proper tile size (`tile` on SW26010 is different from that on conventional architecture, as described in II-B) and unrolling factor for loop tiling and loop unrolling optimizations. Our goal is to prove that the static tuning method can be highly efficient in searching for the optimal code variant. At the same time, the code variant suggested by the performance model is also effective in execution.

We use five programs from the Rodinia benchmark suite that are rich in loops amenable for loop tiling and unrolling. Table II reports the performance of the tuning results, as well as the tuning time. The dynamic tuning needs to run the different versions of the program. The static tuning does not; its tuning time mostly consists of the compilation time. We also measure the performance boost compared with default parameter setting (Speedups in Table II). The compiler used to generate the code variants for both static and dynamic methods is based on the Pluto compiler [12].

The search space consists of multiple dimensions: which loop(s) to tile, which loop(s) to unroll, the tile size of each of the loops to tile, and the unrolling factor of each of the loops to unroll. The complete space can be enormous, considering the combinations of the many loops and the large possible ranges of tile sizes and unrolling factors. Many studies have proposed various ways to prune tuning space [13]–[16]. They are orthogonal to the tuning methods and can benefit both the static and dynamic methods. In our experiments, we keep the search space the same for the two tuners for fair comparisons.

The two tuners find the same transformation parameters on three out of the five programs, and differ slightly on the other two programs `cfd`(4%) and `backprob`(6%) due to some small performance estimation errors. The code variants they pick achieve similar speedups over the original code. On the other hand, the static tuner dramatically reduced the tuning time (by 26X to 43X.) On `lud`, for instance, the tuning time gets shortened from 13.6h to 0.3h. These results indicate the promise of the performance model-based static auto-tuning for guiding compilers in code optimizations on SW26010.

To compare against prior hand-tuned work [17], we further select two kernels (*micro_mg0_1* and *mcica_subcol_lw*) from WRF Physics. According to their methods, the floating-point performance are 421 Gflops and 127 Gflops respectively while our auto-tuning method yields 500 Gflops and 148 Gflops. It is worth noting that this experiment is set up on one CG, of which the peak floating point performance is 765 Gflops. In both methods, the implementations are based on SWACC while our model leads to a better configuration.

Another concern for auto-tuner is input sensitivity. The enabled static tuning employs representative inputs (shown in Table II). Figure 7(b) shows the result of different input domain sizes. Input size does not affect the accuracy of our model. The rationale is, on such a software-cache design, memory access can be precisely modelled although the input size varies.

TABLE II
SPEEDUPS BY TUNING RESULTS AND TUNING TIME SAVINGS BY THE STATIC TUNER.

Kernels	Data Size	Speedups		Tuning Time
		Static	Dynamic	Savings
k-means	3952160*8*32	3.77x	3.77x	41.5x
cfd	193474*4	1.67x	1.74x	40.0x
lud	1600*1600	2.76x	2.76x	43.6x
hotspot	1024*1024*4	2.41x	2.41x	36.2x
backprob	1048576*64	1.84x	1.96x	26.3x

VI. RELATED WORK

A. Performance Models

There have been many previous works on performance modeling [18] [19]. Many of them follow the thoughts of first-order analytical model [20], which predicts the total execution time by summarizing the *missing events*. Chen and others [21] extended the model with considerations of the effects of pending cache hits, data prefetching, and MSHRs. Other work [22] provides insights to the mechanism of superscalar out-of-order processors by paying extra attention to the dispatch efficiency and branch efficiency. In a more recent work [23], the authors try to understand the behaviours of in-order processors by considering the computation resource contention and instruction dependency.

However, they all focus on the fine-grained execution mechanism of cache-based processors and need the runtime profiling statistics. Moreover, memory level parallelism (MLP) was believed very low in [23] but in fact is a key factor in emerging massive parallel architecture. The CPEs of SW26010 access main memory bypassing cache, which makes memory latency and contention the first order considerations of the performance. Our model highlights the memory parallelism and contention among the CPEs.

Recent years have seen increasing interests in performance modeling of many-core processors. Roofline [24] is a coarse-grained performance model for multicore architecture. It analyzes the ratio of the computation operation and memory accesses (arithmetic intensity) and predicts the theoretical peak performance of a program with the "Roofline" curve. A prior work [25] demonstrates the effectiveness of the model for studying the performance bounds of programs on KNL. Roofline model is, however, not designed to precisely predict the execution time of the program – which is the objective of our modelling. For that reason, the subtle effects of some of the optimizations discussed in the paper cannot be captured by upperbound analysis by the Roofline model. For example, when we decrease the DMA request granularity as mentioned in Section IV-1, the arithmetic intensity does not change but the overall performance gains. Another example is that fewer CPEs bring better performance in WRF dynamics cases. Our model explains these phenomena well. Statistics modelling [26] don't need to take much domain knowledge into account, profiles the runs with varying input parameters and uses the regressive method to build performance models. The training takes time and the accuracy is limited by the set of training inputs. There are also models focusing on the CPU cache efficiency of loop structure [27], data placement on various GPU caches [28], the impact of GPU registers/instructions usage on performance upper bound [8]. The work most closely related with our model is an analytical model for GPUs [7]. Two key concepts are introduced: the Memory Warp Parallelism (MWP) and Computation Warp Parallelism (CWP). Their model considered the Memory Level Parallelism (MLP) between warps and the hidden cost through context switching. A follow-up work [6] has improved the MWP-CWP model by considering the MLP within a warp, the SFU contention, the cache effect. Hong and others recently propose the use of abstract kernel emulations and latency/gap modeling of resources to help identify the performance bottlenecks of a GPU program execution, and demonstrate promising results [29].

Our performance model differs from them in some important aspects. First, SW26010 architecture is substantially different from GPUs and the underlying memory access methods are Gload and DMA. The former is subject to waste of memory transactions and the latter long latencies. Second, GPU models focus on memory/computation overlapping in one Stream Multiprocessor (SM) and different SMs are equally treated. We treat CPEs equally only for computation modeling; for modeling memory accesses and memory/computation overlapping, we treat all the CPEs as a whole, which helps accuracy. We further demonstrate the promise of static auto-tuning empowered by the precise performance model.

B. *Auto-tuning*

Auto-tuning typically consists of two components. The first generates code variants in the search/optimization space, and the second assesses the quality of the variants. All the previous studies have relied on actual executions of the variants to do the assessment. To prune the search/optimization space, some effective approaches have been proposed and applied in auto-tuning system, such as evolution (or its variants) methods [13], parallel rank order algorithm [15] and distinct lines model to measure cache line consumption [27]. Our purposed performance model greatly speeds up the assessment process by enabling accurate static predictions of the performance, and hence successfully enables the performance model-based static auto-tuning. It complements the various space pruning methods for accelerating the auto-tuning process.

VII. Conclusions

In conclusion, this work shows that it is feasible to construct a static performance model for a cache-less massive parallel processor SW26010, with an accuracy as high as 95% on average. The model carefully formulates the performance of memory behaviors, including contention and computation-memory overlapping by using a newly-introduced metric MRP. It employs a simple virtual grouping concept to simplify the treatment to massive parallelism, which turns out to be quite effective. It then investigates the opportunities that the precise static performance model creates for analyzing the effects of program optimizations. These usages largely lower the barriers for program optimizations on SW26010, demonstrated by the insights gained on the case studies on DMA granularity, double buffering and active CPE number. Compared with dynamic auto-tuning, our static performance model empowered auto-tuning speeds up the tuning process by as much as a factor of 43, while keeps the tuning quality loss below 6%.

Acknowledgment

The authors would like to thank all the reviewers for their insightful comments and suggestions. This work is partially supported by National Key RD Program of China (Grant No. 2017YFA0604500 and 2016YFA0602100), and National Natural Science Foundation of China (Grant No. 91530323). Xipeng Shen's work is supported by DOE Early Career Award (Grant No. DE-SC0013700), the National Science Foundation (NSF) under Grant No. CCF-1455404, CCF-1525609, CNS-1717425, and CCF-1703487. Xiaomeng Huang's work is supported by National Key RD Program of China (Grant No. 2016YFB0201100), Qingdao National Laboratory for Marine Science and Technology (Grant No. QNLM2016ORP0108), and National Natural Science Foundation of China (Grant No. 41776010). Wei Xue is the corresponding author of this paper (Email: xuewei@tsinghua.edu.cn). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOE or NSF.

References

[1] J. Dongarra, "Report on the sunway taihulight system," *PDF). www. netlib. org. Retrieved June*, vol. 20, 2016.

[2] D. Quinlan, "Rose: Compiler support for object-oriented frameworks," *Parallel Processing Letters*, vol. 10, no. 02n03, pp. 215–226, 2000.

[3] Y. Ao, C. Yang, X. Wang, W. Xue, H. Fu, F. Liu, L. Gan, P. Xu, and W. Ma, "26 pflops stencil computations for atmospheric modeling on sunway taihulight," in *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 2017, pp. 535–544.

[4] H. Fu, J. Liao, W. Xue, L. Wang, D. Chen, L. Gu, J. Xu, N. Ding, X. Wang, C. He *et al.*, "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*. IEEE Press, 2016, p. 83.

[5] C. Yang, W. Xue, H. Fu, H. You, X. Wang, Y. Ao, F. Liu, L. Gan, P. Xu, L. Wang, G. Yang, and W. Zheng, "10m-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 6:1–6:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014912

[6] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A performance analysis framework for identifying potential benefits in gpgpu applications," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 11–22.

[7] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 152–163. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555775

[8] J. Lai and A. Seznec, "Performance upper bound analysis and optimization of sgemm on fermi and kepler gpus," in *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*. IEEE, 2013, pp. 1–10.

[9] NVIDIA Corporation, *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2016.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 44–54.

[11] J. Michalakes, J. Dudhia, D. Gill, T. Henderson, J. Klemp, W. Skamarock, and W. Wang, "The weather research and forecast model: software architecture and performance," in *Proceedings of the 11th ECMWF workshop on the use of high performance computing in meteorology*, 2005, pp. 156–168.

[12] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "Pluto: A practical and fully automatic polyhedral program optimization system," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer, 2008.

[13] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch, "A multi-objective auto-tuning framework for parallel codes," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE, 2012, pp. 1–12.

[14] K. Hoste, A. Georges, and L. Eeckhout, "Automated just-in-time compiler tuning," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '10. New York, NY, USA: ACM, 2010, pp. 62–72. [Online]. Available: http://doi.acm.org/10.1145/1772954.1772965

[15] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth, "A scalable auto-tuning framework for compiler optimization," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12.

[16] A. Tiwari and J. K. Hollingsworth, "Online adaptive code generation and tuning," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 879–892.

[17] H. Fu, J. Liao, N. Ding, X. Duan, L. Gan, Y. Liang, X. Wang, J. Yang, Y. Zheng, W. Liu *et al.*, "Redesigning cam-se for peta-scale climate modeling performance and ultra-high resolution on sunway taihulight," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 1.

[18] P. Michaud and A. Seznec, "Data-flow prescheduling for large instruction windows in out-of-order processors," in *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*. IEEE, 2001, pp. 27–36.

[19] P. Michaud, A. Seznec, and S. Jourdan, "Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors," in *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*. IEEE, 1999, pp. 2–10.

[20] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *ACM SIGARCH Computer Architecture News*, vol. 32, no. 2. IEEE Computer Society, 2004, p. 338.

[21] X. E. Chen and T. M. Aamodt, "Hybrid analytical modeling of pending cache hits, data prefetching, and mshrs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 3, p. 10, 2011.

[22] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Transactions on Computer Systems (TOCS)*, vol. 27, no. 2, p. 3, 2009.

[23] M. B. Breughe, S. Eyerman, and L. Eeckhout, "Mechanistic analytical modeling of superscalar in-order processor performance," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 4, p. 50, 2015.

[24] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[25] D. Doerfler, J. Deslippe, S. Williams, L. Oliker, B. Cook, T. Kurth, M. Lobet, T. Malas, J.-L. Vay, and H. Vincenti, "Applying the roofline performance model to the intel xeon phi knights landing processor," in *International Conference on High Performance Computing*. Springer, 2016, pp. 339–353.

[26] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil, "A predictive performance model for superscalar processors," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 161–170. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2006.6

[27] J. Shirako, K. Sharma, N. Fauzia, L.-N. Pouchet, J. Ramanujam, P. Sadayappan, and V. Sarkar, "Analytical bounds for optimal tile size selection," in *International Conference on Compiler Construction*. Springer, 2012, pp. 101–121.

[28] G. Chen, B. Wu, D. Li, and X. Shen, "Porple: An extensible optimizer for portable data placement on gpu," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 88–100. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.20

[29] C. Hong, A. Sukumaran-Rajam, J. Kim, P. Rawat, S. Krishnamoorthy, L. Pouchet, F. Rastello, and P. Sadayappan, "Gpu code optimization using abstract kernel emulation and sensitivity analysis," in *Proceedings of the 39th annual ACM SIGPLAN conference on Programming Language Design and Implementation, 2018. (PLDI 18)*, 2018.