修改稿

PRF：A Process-RAM-Feedback Performance Model to Reveal Bottlenecks and Propose Optimizations①

Xie Zhen(谢震)[*#], Tan Guangming[*], Liu Weifeng[^], Sun Ninghui②[*]

([*]Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

([#]University of Chinese Academy of Sciences, Beijing 100190)

([^]College of Information Science and Engineering, China University of Petroleum, Beijing 102249)

Abstract

Performance models provide insightful perspectives to allow us to predict performance and to propose optimization guidance. Although there has been much researches, pinpointing bottlenecks of various memory access patterns and reaching high accurate prediction of both regular and irregular programs on various hardware configurations are still not trivial. In this work, we propose a novel model called Process-RAM-Feedback (PRF) to quantify the overhead of computation and data transmission time on general-purpose multi-core processors. The PRF model predicts the cost of instruction for single-core by a DAG (Directed Acyclic Graph) and the transmission time of memory access between each memory hierarchy through a newly designed cache simulator. By using performance modeling and feedback optimization method, we use PRF model to analyze and optimize convolution, sparse matrix-vector multiplication and sn-sweep as case study for covering with typical regular kernel to irregular and data dependence. Through the PRF model, we obtain optimization guidances with various sparsity structures, algorithm designs, instruction sets support on different data sizes.

Key words: performance model, feedback optimization, convolution, sparse matrix-vector multiplication, sn-sweep

0 Introduce

In recent decades, modeling has been employed to optimize performance of computational kernel and verify the validity of proposed optimization methods[1,2,3,4]. Depending on whether the model uses hardware and software features as a reference, it can be roughly divided into two categories. One is "black box" model that uses a fitting or machine learning method to predict the performance by extracting characteristics of the target machine and collecting data of application. This method collects a large amount of application data to build a model by statistics or mathematical model. The model is not universal and does not reflect the real implementation inside architecture. The other is "white box" model, which uses a simplified machine model to describe the matching relationship between application and hardware. The simplest white-box model is the Roofline model[5] proposed by Williams et al., which can be used to bound floating-point performance with a function of machine peak performance, peak bandwidth and arithmetic intensity. However, the Roofline model cannot describe detailed bottlenecks beyond memory bandwidth and peak performance. It only show the upper limits of the performance. A more detailed machine model with memory hierarchy is the Execution-Cache-Memory (ECM) model[6] proposed by Holger et al.. Based on ECM model, stencil[7] loop kernel[8] and microbenchmark[9,10] have been modeled and optimized. It divides the machine into in-core and out-core phase, and reflects the time of instruction execution on processor core and data transfer between cache and memory by an address-based cache simulator (Pycachesim) or proportion analysis by Kerncraft[11]. But mapping real application to address sequence requires a medium and none of these tools have yet been provided. Also this model ignores the impact for possible occurrence of data dependence and the differences between regular and irregular memory access. So it cannot accurately predict performance when the application has pipeline stall and stride or random memory access. Therefore, it cannot give specific feedback based on the incomplete information. Moreover, for specific applications, optimization methods are very different with different instructions and data sizes, the existing models cannot give fine-grained optimization guidance. In addition, there have been many work based on hardware performance counters[12,13,14,15]. Although these methods can also obtain the load characteristics by the number of monitored events, it needs to monitor at runtime with unacceptable overhead.

In this paper, we propose a new performance model called Process-RAM-Feedback. It deeply considers the diversities of the pipeline and the cache layers as a "white box" model. So that it can predict the instruction overhead and the cache misses for regular, irregular and data dependence applications. In this sense, our model largely broadens the application domain of performance modeling. More importantly, our model can give feedback guidance for the optimization, which is not available in existing performance models. In order to realize this functionality, our model consists of three fundamental steps. Firstly, our model abstracts the important hardware parameters, such as the calculation unit, access unit, instruction cost, the associativity and the size of each memory level (cache and DRAM). Secondly, based on these parameters, we construct a DAG to predict

① Supported by the National Key Research and Development Program of China and the National Natural Science Foundation of China.

② He was born in 1968. Professor and Supervisor for Ph.D. students with University of Chinese Academy of Sciences. Fellow Member of China Computer Federation. E-mail: snh@ict.ac.cn

instruction overhead and a cache simulator to predict the cache miss as the overhead of data transmission. Lastly, our model can use the obtained information in the previous two steps, reveal the bottlenecks, compare the effects of different optimization methods, and then provide optimization guidances. In this way, the optimized kernels are obtained. Our main contribution is reflected in three aspects:

1. In the proposed PRF model, we consider comprehensive factors which may effect the kernel performance, e.g., the probability of instruction pipeline stall and cache line transmission between different levels of RAM, and the execution time of instructions both calculation and memory access. Based on the theoretical and model-based analysis for specific cases, our model can produce a much more accurate performance prediction and extend the application domain to the context of irregular memory access and data dependency issues.
2. In order to get the number of cache misses at all cache layers for irregular case, we develop a multi-level cache simulator which can be easily built on the target hardware and quickly get the needed information for the specific kernels. As an indicator of data transmission, improvement of prediction accuracy for cache miss is a crucial factor which affects the accuracy of performance prediction in our model.
3. According to the different performance bottlenecks, our model feeds back developers to select the best optimization method and informs performance expectation with various data inputs, instructions support and data sizes.

From regular memory access to irregular and data dependence, we select three cases (convolution, Sparse matrix-vector multiplication and sn-sweep) to verify the correctness of the model. Our experiments show that, for modeling the convolution algorithm as the regular case, the PRF model could feed back the best optimization suggestion for various data sizes under current instruction support. For the irregular memory access, by running SpMV with 3673 sparse matrices from the Florida Collection[16], the average error rate : ABS(predicted_value - measured_value) / measured_value of our PRF model is about quarter than the ECM model. Further, for our sparse matrices, feedback optimization guides developer to select the parameter (i.e., block size) for the Block CSR sparse matrix format by using the output of the PRF model and it highly matches the best block size manually selected from exhaustive experiments. As for sn-sweep, PRF model constructs the DAG through its calculation process, and analyzes the data dependency relationship between instructions to reveal the problem of pipeline with out-of-order execution support, and finally guides to solve data dependence issue and achieves performance improvement is about 192% in single-core pipeline with linear scalability for multi-core.

1 Related Work

Performance modeling is a very useful technique for optimization of parallel applications on HPC platforms. All of the current architecture can be divided into the instruction part and the memory part. As shown in Figure 1, it results different abstract methods can build various performance models. Next we begin to introduce the differences between four performance models.

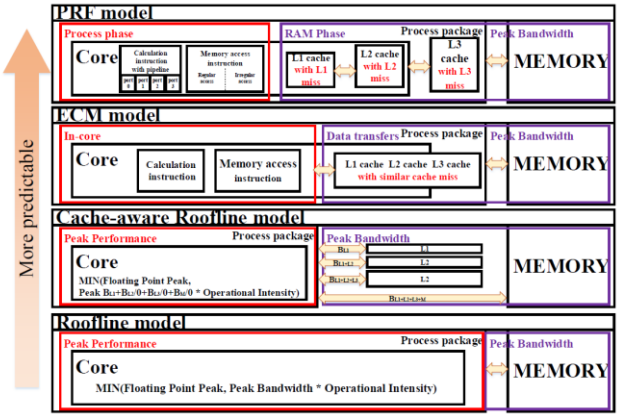

Figure 1: PRF model and comparison with Roofline, Cache-aware Roofline and ECM

The Roofline[5] Model is a visual analytical model used to pinpoint performance bottlenecks. For the instruction part, the Roofline model abstracts it as a black box, this model describes the peak computing performance as the upper limit of the instruction part, which constraints the best performance which program can achieve. Same as before, the Roofline model also abstracts memory part as the peak memory bandwidth. It can also be applied to any program. However, the model gives little detail information, and lacks accuracy of performance prediction for target application.

As a refined model on Roofline, Cache-aware Roofline[17] joins the implications of the cache hierarchy, and the transmission bandwidths of cache are used as boundaries. But this model has the same disadvantages as Roofline by low accuracy.

The ECM[6] model considers the time for executing the instructions with data coming from the L1 cache as well as the time for moving the required cache lines (CLs) through the cache hierarchy. It also calculates the time for executing instructions of loop kernel on the processor core (assuming no cache miss) and transferring data between its initial location and the L1 cache. The in-core execution time T_core is determined by the unit that takes the most cycles to execute the instructions. The time needed for all data transfers required to execute one work unit is expressed as T_trans. The in-core execution and transfer costs must be put together to arrive at a prediction of single-thread execution time. By simulating execution of CPU instruction, the model has a good prediction accuracy for regular memory access pattern. But ECM could predict performance in the memory part of a sort, it divides memory part into an undifferentiated cache layer and main memory, resulting in hard to model data locality.

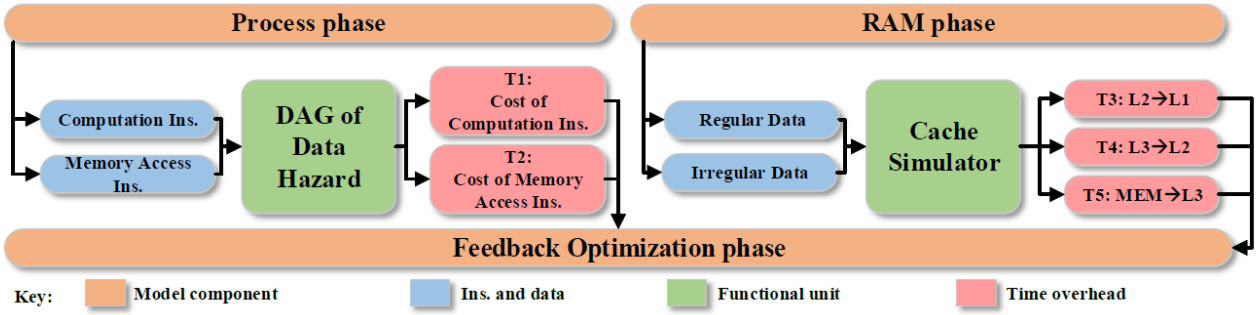

Figure 2: Three components and workflow of the PRF model. PRF divides model into three parts: "Process phase", "RAM phase" and "Feedback optimization phase". The "Process phase" corresponds to the instruction part, which predicts the cost cycles within the CPU core by executing internal instruction with pipeline, while the "RAM phase" corresponds to the memory part, it describes the connected relationship and predicts the time between memory hierarchy. The "Feedback optimization phase" collects the output of each phase, analyzes execution bottlenecks and potential opportunities, and provided developer optimization guidance.

Therefore, for irregular application, these three models are difficult to give accurate prediction, and the error mainly occurs in the memory access phase, and there is a considerable gap between the size of data transfer and the data reuse of irregular application by these model predicts and real situation. Also the ECM model ignores the effect of pipeline stall. This is also the main focus of our model in this paper. Here we realize a pipeline DAG with more realistic memory hierarchy, and design a cache simulator to output approximately real cache miss for irregular case. With our abstract method, our PRF model can achieve better predictive accuracy, while also covers more irregular and data dependence cases. The comparison of all models is shown in Figure 1.

Cache simulation has been used to evaluate memory systems for decades. Khatwal and Jain[18] examined the topic for multiprocessor memory-systems, Hui et al. [19] investigated an x86 cache simulation framework in the specific context of multiprocessor systems, Mohammad[20] discussed many intersection properties for caches using different replacement policies, and Somdip[21] study coherency protocols between different caches in a shared memory system. These simulators are constructed by configurations, such as cache size, sets, cache associativity, block size, replacement policy according to a specific application. However, for program optimization, these cache or memory simulators are not efficient enough since the cost for lots of hardware emulation functions. Thus applying them in real-world applications will lead to new performance bottlenecks. In this paper, we design a light weight cache simulator that can simulate all levels of cache miss with a very low overhead for all memory access patterns.

As for regular memory access pattern, convolution operation in convolution neural networks is the most time-consuming function. Qadeer et al. [22,23] use the vectorization instruction or design the unique hardware structure to speed up the calculation, but these optimization are case by case, since the effective optimization method for different data sizes and machine structures is quite different, so our model is to find a set of methods to optimize the regular memory access applications in different architectures and guides the developer to obtain higher performance.

Sparse matrix-vector multiplication (SpMV, y=Ax) is an important computational kernel with an irregular memory access patterns. The SpMV operation does not contain any dependencies thus has relatively high parallelism. Buttari et al. [24] uses a modeling method to optimize the SpMV by blocking. But this model fits machine characteristics and predicts performance by fill-in the dense matrix to achieve the best block sizes. Vuduc et al. [25] uses register-level tiling opportunities to select parameters for BCSR by model. However, these parameter tuning is time-consuming and none of these methods can reflect real execution on the current architecture of SpMV. Then we quantify instruction and memory transmission for various parameter, and select the best block size and predict optimized performance.

Sn-sweep is a typical regular application with very low performance as data dependence. It is the largest time-consuming function for numerical simulation of radiation transport in high energy density plasma physics[26]. As sn-sweep can be seen sweep the radiation flux from the source across the grid in the downstream direction, so grid decomposition method makes task parallelization easier. Jie[27] describe a message passing implementations of sn-sweep algorithms within a radiation transport package to increase expandability. Our model focuses on bottlenecks within a task and expose the cause of the instruction pipeline stall to guide optimization.

2 The PRF Performance Model

Our PRF (Process-RAM-Feedback) divides model into three parts: "Process phase", "RAM phase" and "Feedback optimization phase". In Figure 2, we visualize these three phases, and set the time of "Process phase" when all instruction and data are in the L1 cache, the time of "RAM phase" is the transfer time between the memory hierarchy. The output of model are evaluated, analyzed and directed by "Feedback optimization phase".

For "Process phase", we first divide the instruction into computation instruction and memory access instruction, and by building a DAG (Directed Acyclic Graph) to describe data dependencies. Through the implementation of the pipeline, it output the overhead for two kinds of instructions. Then we label the cost time of calculation and memory access instruction as T1 and T2. For the "RAM phase", due to the principle of locality, different inputs and applications will result in different cache miss, so memory access instruction does not definitely cause data transfer (if data are in cache). Also we found that the current Intel processors have a prefetching mechanism for the regular memory access, resulting in almost no cache miss at some cache levels. So we separate the regular and irregular data, and the time of transmission for regular data can be calculated by format which will be described in detail later, and also build a multi-level cache simulator for irregular data, then simulator can output the number of cache misses at all levels, therefore we can obtain the time of all the data transfer and label the transmission time of three-level cache as T3, T4, and T5. The "Feedback optimization phase" abstracts out four performance bottlenecks, uses the output of the model to point out the key factors which affects performance most, and guides developer improve performance which optimization can bring.

2.1 Process phase

Instruction are mainly divided into two types: computation (addition and multiplication) and memory access (load and store) instruction. The two kinds of instructions are scheduled independently in core internal and can properly implement instruction-level parallelism without data dependence. Although current processor has out-of-order execution mechanism, when data dependency occurs, the instruction which depends on the previous results will also causes performance stall[28].

In order to model the pipeline, we design a DAG module, which is constructed as shown in the Figure 3, it uses a blank circle representing read without write, a solid line with an arrow representing the direction and load a data, and the grid circle representing read after write. By analyzing the code of an application to build the DAG, execution flow of the DAG shows pipeline analysis of data dependent. For example, on a three-stage pipeline processor, R3 occurs data dependence for read after write, resulting in one stall for data dependence.

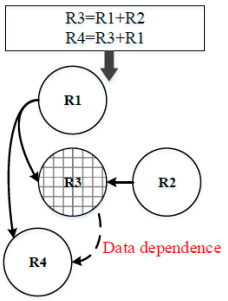

Figure 3: DAG for modeling data dependence

As an example, we build PRF model on an Intel Haswell architecture. This microarchitecture can execute one addition and one multiplication operation or two FMA operation per cycle, while supporting two load operations and one store operation per cycle. All details will be introduced in the Section 3. So if the numbers of addition, multiplication, FMA, load and store instruction are A, M, FMA, L and S respectively for an application, then the time spent on "Process phase" can be calculated by Formula 1.

$$T(Processphase) = \mathbf{MAX}(\mathbf{DAG}(A, M, \frac{FMA}{2}), \mathbf{DAG}(\frac{L}{2}, S)) \quad (1)$$

2.2 RAM phase

"RAM phase" converts cost of data transmission into overhead between each level of cache and main memory. As the size of each level of memory hierarchies are very different for various architectures, data may appear in any of them, resulting in complicated data transfer time. When a CPU requests memory access, it will seek a data block as cache line from L1, L2 and L3 caches to main memory in turn. If the former search is missing, this cache line will transfer from the lower level of memory hierarchy to the upper level. At the same time, the current cache has a perfect prefetching mechanism for the regular memory access, resulting in a significant reduction in transmission time.

Therefore, as in Figure 2, "RAM phase" model partitions memory accesses into regular and irregular memory access. For regular memory access, it is to access continuous memory address or cache line, or visit fixed stride memory address. For the Haswell core we used to model, the L1 cache cannot implement prefetching operation as the first level of cache by lots of experiments. Then for other cache levels, when data is reading from L2 cache to L1 cache, the adjacent data which will be accessed is transferring from L3 cache to L2 cache, and since the transmission bandwidth of L3 to L2 is half than L2 to L1, it leads to a half overlap. At the same time, other adjacent data which will be accessed is transferring from the main memory to L3 cache, and the bandwidth of main memory to L3 is lower than the cache, so there will be a delay by the differences.

Assuming that the internal bandwidth of cache which data appear is C, the bandwidth between the main memory and the last level cache is M, and the regular data size to be accessed is Amount_regular, the time spent on regular memory access pattern is given in Formula 2:

$$T(RAM\_phase\_for\_regular) = \frac{Amount\_regular}{Amount\_regular \le Last\_Cache\_Size ? C : M} \quad (2)$$

For irregular memory access, cache prefetcher cannot work well when access disordered address, the data which needs to be accessed may appear on any of the memory hierarchy, so loading these irregular data will produce unexpected cache misses. In order to get these cache misses, we design a cache simulator, which could simulate cache groups and cache lines, and use index number (similar to the address) to distinguish each cache line. Then the simulator builds memory access sequence based on the user's input data and memory access process, and finally outputs cache misses. It also simulates the replacement mechanism.

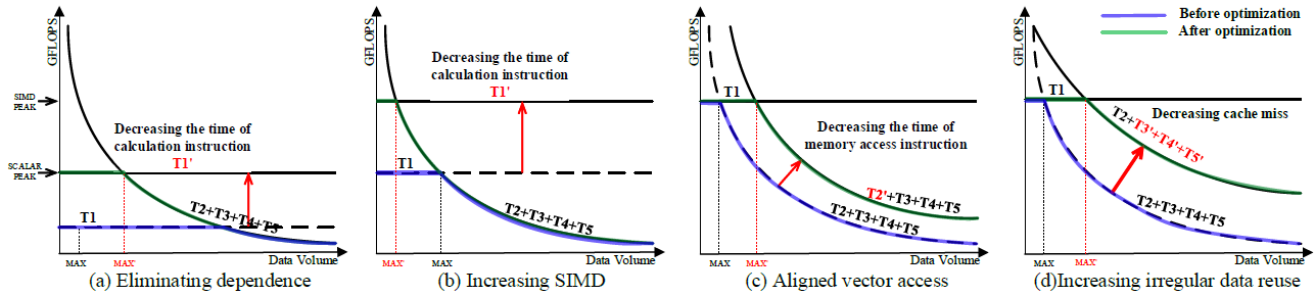

Figure 4: Four different optimization opportunities.

Unfortunately, the Intel smart cache replacement policy is confidential, so we carry out a large number of experiments and obtain some conclusions: L1 cache use LRU (Least Recently Used) policy, L2 and L3 are based on the LRU policy with relatively perfect prefetching operation for regular memory access.

Therefore, the process of building the cache simulator can be described as follows: 1. Detect the size and group associative of the cache at all levels, and assign corresponding tags which could mark block number and valid bit. 2. Build the mapping relationship at all levels. 3. Set its replacement strategy.

The internal work-flow of the cache simulator is as follows: 1. Detect current physical machine and build the cache simulator. 2. Partition the input data which will be accessed to corresponding cache lines and mark it as regular or irregular. 3. The cache simulator reads cache lines by marking the cache block valid nor not, prefetchs regular cache lines and records the miss numbers. 4. Finally, through the known data transfer rate, the cache simulator will output cache miss and cost at any of the memory hierarchy. If the three cache misses is L1_miss, L2_miss and L3_miss respectively and the size of Cache Line is CL, the time spent is given in Formula 3:

$$T(RAM\_phase\_for\_irregular) = \frac{(L1\_miss + L2\_miss) * CL}{C} + \frac{(L3\_miss) * CL}{M} \quad (3)$$

The sum of Process part and RAM part shows the overall completion time of the program. It can be calculated by Formula 4. And GFLOPS can be calculated by Formula 5.

$$T(PRF) = T(Process\_phase + RAM\_phase - overlap)$$
$$= MAX(DAG(A, M, \frac{FMA}{2}), DAG(\frac{L}{2}, S) + T(RAM\_phase))$$
$$= MAX(DAG(A, M, \frac{FMA}{2}), DAG(\frac{L}{2}, S) + T(RAM\_for\_regular) + T(RAM\_for\_irregular)) \quad (4)$$

$$GFLOPS(PRF) = (A + M) * CPU\_Frequency / T(PRF) \quad (5)$$

2.3 Feedback optimization phase

According to Figure 2 and Formula 4, we split the time for each stage is T1 to T5.As shown in Figure 4, the abscissa indicates the data size, and the ordinate indicates the GFLOPS. The Line T1 represents floating-point performance without memory access time, and the line (T2+T3+T4+T5) represents floating-point performance without calculation time. So the final execution time is the minimum value of line T1 and line (T2+T3+T4+T5). We enumerate four different possibilities and show us how to discover the bottleneck and optimize our code.

As shown in Figure 4(a), it can be observed the line T1 cannot reach scalar peak performance without data transmission, so feedback optimization guide developer to choice intermediate variable to reduce the data dependence for addition and multiplication operation. Then the red arrow represents the reduction of T1 and improvement of GFLOPS by decreasing the time of calculation instruction. Through the changes of feedback optimization, the purple performance line can be upgraded to the green performance line with the data size increases. As same for Figure 4(b), the line T1 cannot reach floating point peak performance without data transmission, so feedback optimization guide developer to choice the SIMD to increase calculation speed. If T2 has spent much more time and become a bottleneck, it presents the data access instruction has become worthwhile optimization. Feedback optimization informs developer to redesign data structure or increase the redundant space to reduce the time of memory access instruction, and the optimization effect is shown in the Figure 4(c). For If we find that the memory access part is bottleneck, then feedback optimization will warm developer to increase cache utilization and reduce the transmission time by using new data format or cache-based block design in the Figure 4(d).

Next, we will use three cases to implement the whole process of modeling.

3 Experimental Testbed

We use an Intel Xeon E5-2680 v3 processor[29] with the SSE, AVX and AVX2 support for validating our model. Each core can execute one multiply and one addition in floating point or two FMA instructions per cycle without data dependency. The memory hierarchy which consists of three on-chip SRAM data caches. For the scalar instruction[30], the memory read and write instruction (load & store) is only one kind, and the data is always aligned. For the vector instruction, memory access instruction is divided into load, loadu, store and storeu by respectively accessing aligned and unaligned data, and the time spend on unaligned data is twice as large as for aligned data. All the specific specifications see Table 1.

Table 1: Special machine parameters

Machine Parameters	Details
CPU	Intel Xeon E5-2680 V3 12 cores/24 threads 2.5GHz with Turbo Mode off
Cache	L1 : 8way 32*12KB L2 : 8way 256*12KB L3 : 20way 30MB
Cache Line, FPU width	64Byte, 2*256 Bit FMA
L1D Bandwidth/cycle	2*32 Byte load + 32 Byte store
L2/L3 Bandwidth/cycle	64 Byte/32 Byte
Memory Channels	4-channel DDR3-1866 up to 42.6GB/s
Compiler/Compile Options	Intel icc 15.0.2/ -opt-prefetch=3 -O3

4 Validation

4.1 Convolution

4.1.1 Convolution operation

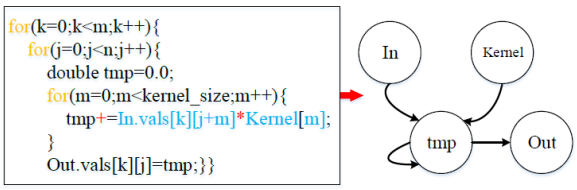

Figure 5: DAG of 1D convolution

Convolution operations have been widely used in denoising, extraction, structure smoothing, filtering, detection, image enhancement and many other image processing applications. The information of images is encoded in the spatial domain rather than the frequency domain, thus the image convolution operation is extremely essential and useful in image processing. For real-world applications, 32-bit floating point is usually selected for rapid training. For 1D convolution, the convolution filter is a 1-dimension structure, as horizontal filter with size of 1*N. 1D convolution operation simply rotates the convolution kernel 180 degrees before multiplying the input data. Figure 5 illustrates an m*n input data convolved with a 1*16 kernel size and its DAG. Each pixel in the window is multiplied by their corresponding kernel coefficients and finally generate the whole output data.

4.1.2 Performance prediction for naive code

From our example, it can be seen that the 1D convolution requires reading 16 kernel data and 16 input data to perform 16 addition and multiplication calculations. We can easily find out data is reuse access, along one dimension for 1D convolution. The next iteration operation will reuse 15 elements from the previous iterations.

For this program, the KERNEL array has good locality, and it can always be stored in L1 cache. Reading a cache line of IN and OUT data separately will updates 16 values of OUT, and contains both 16*16 addition and multiplication instructions, and therefore it also produces 16*16 load and 16 store instructions. For the current architecture, each cycle can execute an add and mul instruction, so the cost cycle of all of those addition and multiplication instructions is: 16*16=256, and a cycle can perform two load instructions and one store instruction. So the cost cycle of access instruction is: 256/2=128, For L2->L1 and L3->L2, a cycle can transfer one cache line and half of a cache line respectively, namely data transmission requires one or two cycles for a cache line, For MEM->L3, a cycle can transmit 1/5 cache line, finally we can infer the GFLOPS performance in different data sizes are shown in Table 2, for example, the GFLOSP = (16*16*2)/(max(256,128+2/4/10)/2.7) Gflops:

Table 2: GFLOPS performance in different data sizes for naive code, and comparison with PRF, ECM and measured.

	L1	L2	L3	MEM
T1 : add&mul	256	256	256	256
T2 : load&store	128	128	128	128
T3/T4/T5 : memory hierarchy	0/0/0	2/0/0	2/4/0	2/4/10
prediction GFLOPS by PRF	5.4	5.4	5.4	5.4
prediction GFLOPS by ECM	5.4	5.4	5.4	5.4
measured GFLOPS	5.22	5.28	5.28	5.27

From the performance analysis given above, it can be seen that, regardless of the data at any memory hierarchy, the largest cost of a 1D convolution is the computational instruction (T1). Next, we will introduce different optimization methods, and analyze the optimized performance.

4.1.3 Optimization method and modeling analysis

By comparing the overhead of each phase, we can see that the T1 becomes a bottleneck. From the Figure 4, the performance achieve the scalar peak which means there is no data dependence in the calculation process, so the optimal GFLOPS can be achieved when using SIMD by "Feedback optimization phase" in Figure 4(b). For the current machine platform, it supports SSE for 128-bit and AVX2 for 256-bit. So we will implement optimized version for AVX2 instruction with unaligned data and give our performance prediction. The pseudo-code is given in Algorithm 1:

Algorithm 1 AVX2 Unroll Unaligned

Input:IN[],length,KERNEL[],kernel_length;
Output:OUT[]
Input:IN[],length,KERNEL[],kernel_length;
Output:OUT[]

1: $\_m256\ kernel\_reverse[kernel\_length](aligned)$;
2: **for** $i = 0; i < kernel\_length; i++$ **do**
3: $kernel\_reverse[i] \leftarrow \_mm256\_broadcast\_ss$
4: $(\&KERNEL[kernel\_length - i - 1])$;
5: **end for**
6: **for** $i = 0; i < (length - kernel\_length + 1); i+ = 16$ **do**
7: $acc0, acc1 \leftarrow \_mm256\_setzero\_ps()$;
8: **for** $k = 0; k < kernel\_length; k+ = 16$ **do**
9: $data\_offset \leftarrow i + k$;
10: **for** $l = 0; l < 4; l++$ **do**
11: **for** $m = 0; m < 16; m+ = 4$ **do**
12: $data\_block \leftarrow \_mm256\_loadu\_ps(in+data\_offset+l+m)$;
13: $acc0 \leftarrow \_mm256\_fmadd\_ps(kernel\_reverse[k + l + m], data\_block, acc0)$;
14: $data\_block \leftarrow \_mm256\_loadu\_ps(in + data\_offset + l + m + 8)$;
15: $acc1 \leftarrow \_mm256\_fmadd\_ps(kernel\_reverse[k + l + m], data\_block, acc1)$;
16: **end for**
17: **end for**
18: **end for**
19: $\_mm\_storeu_ps(out + i, acc)$;
20: **end for**

For 16 iterations of the program, it will produce 16*2 addition and 16*2 multiplication instructions. It also produces 16*2 loadu and 2 storeu instructions and 7 additional addition operations. Therefore, the cost cycle of all addition and

multiplication instructions is: 16*2+7=39. By using AVX2 instruction and data is unaligned, the cost cycles of access instruction is: 16*2*2/2=32. The GFLOPS performance is given in Table 3:

Table 3: GFLOPS performance in different data sizes for naive code, and comparison with PRF, ECM and measured.

	L1	L2	L3	MEM
T1 : add&mul	39	39	39	39
T2 : load&store	32	32	32	32
T3/T4/T5 : memory hierarchy	0/0/0	2/0/0	2/4/0	2/4/10
prediction GFLOPS by PRF	35.45	35.45	35.45	28.80
prediction GFLOPS by ECM	35.45	35.45	35.45	28.80
measured GFLOPS	34.48	34.27	34.28	27.53

4.1.4 Optimization guidance

After the previous experiments, we find that the best optimization scheme depends on the supported instruction set and size of the image data set. From Figure 6, we extend to predict performance from SSE to AVX2 with aligned memory access or not for different data sizes. Then we can clearly observe that the SSE instruction set can achieve very good scalability. When using the AVX2 instruction set, data transmission begin to affect performance with the increase of data size, then there is no way to achieve the peak of floating-point calculation, therefore, using AVX2 instruction set cannot linearly improve performance. As aligned memory access need addition data storage, the method is only efficient with a small amount of data size. So the feedback guideline recommend developer to use vector instruction with unaligned access and increase the memory transmission optimization with AVX2 support.

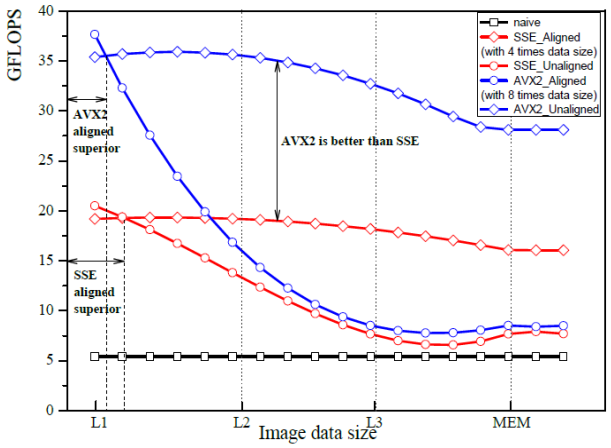

Figure 6: Modeling and prediction of various instruction support and date sizes

4.2 SpMV

In this section, we will use 3627 matrices in the Florida sparse matrix library[16] to verify the PRF model and compare with ECM model (as ECM releases Kerncraft and Pycachesim, so we could only use these two tools and the idea of the ECM paper as much as possible to model SpMV), find bottlenecks and propose optimization scheme. Finally, we choose the optimal implementation with the some randomly selected matrices.

4.2.1 Test matrices

The sparse matrices we used to model cover all the Florida sparse matrix library. And these matrices come from a wide variety of applications with different sparse distribution characteristics. Similar to earlier work on SpMV, our CSR kernel also use row decomposition, loop unrolling and software prefetching. Some randomly selected matrices are used in previous papers[31], and the "bar" matrix is extracted from a real case of our actual problem. In real-world applications, 64-bit floating point is usually selected for better precision.

4.2.2 Performance prediction and bottlenecks

The pseudo-code for CSR-based SpMV is given in the left of Figure 7. For "Process phase", we build data dependence DAG on this pseudo-code, and by using multiple registers, compiler can generate independence code as shown in the right of Figure 7. Then almost all instructions can execute pipeline.

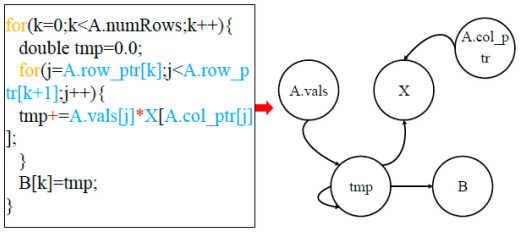

Figure 7: Pseudo-code and DAG for CSR-based SpMV

For a sparse matrix, its row, column and nnz is R, C and NNZ, then the size of integer array A.row_ptr is (R+1)*4 Byte, the size of integer array A.col_index is NNZ*4 Byte, the size of double array A.value is NNZ*8 Byte, the size of the double array for vector X is C*8 Byte. The red place in pseudo-code is multiplication and addition, respectively, there are NNZ ADD and NNZ MUL instructions, while the blue place indicates memory access, A.row_ptr needs (R+1) LOAD instructions, A.col_index needs NNZ LOAD instructions, A.value needs NNZ LOAD instructions, vector X needs NNZ LOAD instructions, the output vector B needs R STORE instructions, it has a total of NNZ ADD, NNZ MUL, 3*NNZ LOAD and (R+1) STORE instructions. Meanwhile, all instructions can execute pipeline. By Formula 1, the time spent on calculation units is NNZ cycles and the time spent on memory access units is 1.5*NNZ cycles. The floating-point operations is NNZ(ADD) +NNZ(MUL)=2*NNZ.

For "RAM phase", the main uncertain time cost needs to model is varieties data transmission of vector X for kinds of sparse matrices. Then, we divide the data of matrix and vectors by the size of cache line and mark as regular and irregular respectively. According to the RAM phase mentioned in Section 2.2, we build a cache simulator by reading hardware parameters of the target machine, such as cache sizes and group associations. The simulator reads the marked data blocks in turn and records the cache misses. Therefore it simulates regular memory access of the CSR matrix with prefetching and out-of-order access of the vector.

The GFLOPS of a special matrix is finally obtained by Formula 5. As shown in Figure 8, the abscissa of all sub-graphs represents the degree of sparsity of the matrix, and it is sorted

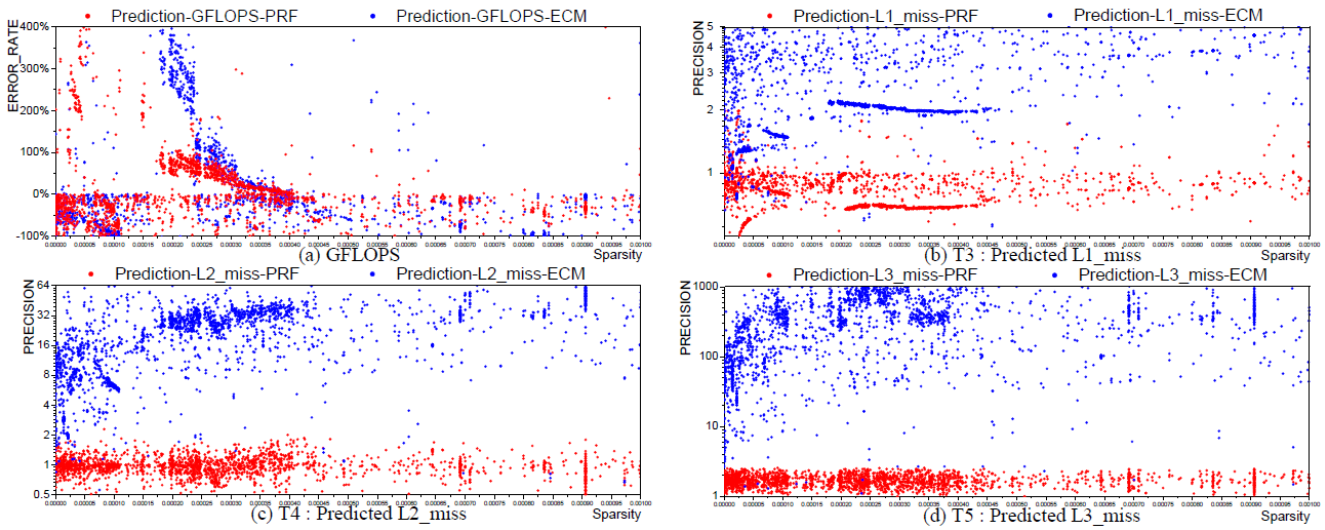

Figure 8: Single core PRF model for SpMV on 3627 sparse matrices. Note that (b)–(d) are in logarithmic scale. The abscissa of all sub-graphs represents the degree of sparsity of the matrix, and it is sorted from small to large in order. The ordinate of (a) represents the error rate (ABS(predicted_value - measured_value) / measured_value). The ordinate of (b)-(d) represent the precision (simulation_time /measured_time).

from small to large in order, and the ordinate of (a) represents the error rate of simulated values to measured data for GFLOPS (giga floating-point operations per second). When the real value is the same as the predicted value, the error rate is 0%, so the neighbourhood of 0% represents that this model can produce fairly accuracy prediction. In contrast, the PRF model is marked with red, and EMC model is blue. The ordinate of (b), (c) and (d) represent the ratio of simulation to measured data for L1 miss, L2 miss and L3 miss and the rate of 1 represents perfect prediction. Our measured L1, L2, and L3 cache misses used PAPI's[32] statistics.

Through our observations, for Figure 8(a), when the sparsity is less than 0.0002 or greater than 0.00045, about half of our points appears between 0.5 and 1.5 which is slightly better than ECM. When the sparsity is between 0.0002 and 0.00045, our predictions are significantly better than the ECM model, and the error rate of prediction is more than twice as small as the ECM. For sub-graph b, c and d, it is found that most of our prediction data are concentrated near the rate of 1, which is obviously contrasted with ECM, and it is also the main reason why our GFLOPS prediction accuracy is better than ECM. But there are several reasons for our predictions not to exactly agree with measurement: 1) the real cache replacement policy is unknown. 2) the actual throughput is not a constant. However, our method can also achieve very high prediction accuracy with Formula 5 and cache simulator.

4.2.3 Feedback performance optimization

Analyze the performance of SpMV, we found that the max time and the bottleneck is (T3+T4+T5). As the CSR format matrix results in a large number of repeated data transmission of slice vector X. By changing the format of sparse matrix, we can increase locality of vector X and reduce matrix index transmission, thus it can reduce the time of data transmission and increase the efficiency of floating-point operation.

So we randomly select 12 matrices and one of our engineering matrices - "bar". Through modeling, we find that the CSR format causes a high L1 and L2 cache miss, then T3 is about 1.8 times than T4 and T4 is about 3.9 times than T5, resulting in a large time of data transfer. So we think of ways to optimize (T3+T4).

For many solutions for cache optimization, register blocking is a typical technique for improving data reuse. The sparse matrix is logically divided into blocks and those blocks usually contain at least one non-zero. SpMV computation proceeds block-by-block. For each block, we can reuse the corresponding elements of the vector X by keeping them in registers to increase temporal locality. Register blocking uses the blocked variant of compressed sparse row storage format and it is also called BCSR for short. Blocks within the same block row are stored consecutively, and the elements of each block are stored consecutively in row-major order. BCSR potentially stores fewer column indices than CSR (one per block instead of one per non-zero). The effect is to reduce memory traffic by reducing index storage overhead and reusing the vector slice. Then the T3 and T4 and even T5 can be reduced. However, a uniform block size may require filling in explicit zero values, resulting in extra computations and data traffic. Based on the above principle, our feedback optimization is implemented based on BCSR format.

Now, we apply the PRF model to the BCSR format. By reading the matrix to cache simulator, we can get the cache misses and increased zero elements calculation, and finial put forward the optimal block shapes, then the partitioning scheme is given. In Figure 9, we model BCSR format for selected 18 matrices, give recommendation with an average speed-up of

173.79% and compare with hand tunning for BCSR. For the "rail4284" matrix, modeling result find that the matrix does not have blocking characteristic, so traditional CSR can achieve better performance. Compared to the direct select optimal parameter by 64 SpMV time, our method greatly saves the overhead of selecting the optimal parameter by 12 SpMV time, so the model greatly improves the efficiency of feedback optimization.

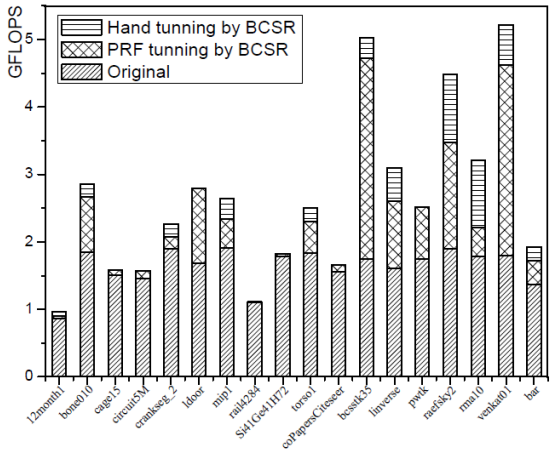

Figure 9: Overall speedup of test matrices by optimization suggestion of PRF model and hand tunning by BCSR format

4.3 Sn-sweep

4.3.1 Sn-sweep operation

In particle transport simulations, radiation effects are often described by the discrete ordinates (Sn) form of Boltzmann equation. In each ordinate direction, the solution is computed by sweeping the radiation flux across the grid. Sn-sweep operations have been widely used in radiation transport, radiation effect and many other high energy density plasma physics applications. For scanning algorithm, sn-sweep is an essential example of calculating the relationship between the influence of adjacent elements. The main process flow of sn-sweep is sequentially calculate the influence of neighbouring elements and update the surrounding elements. The left side of Figure 10 shows a core function of sn-sweep, it needs to multiply the left and upper elements by two weights and add the sum to the current element, then one iteration is to complete corresponding calculation of all the mesh.

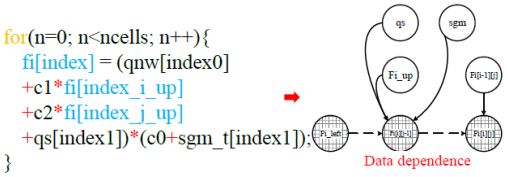

Figure 10: Pseudo-code and DAG for sn-sweep

4.3.2 Performance prediction and bottleneck analysis

In the "Process phase", the pseudo-code is transformed into a DAG, and it be calculated by the line order in each iteration. We can see that when calculating the next element, the result of the previous elements must be calculated. This will lead to the pipeline stall. Fortunately, the Out-of-Order Engine component of the current processor can only mitigate the restrict of data dependency to some extent. For this case, the calculation of an element requires 4 addition, 3 multiplication, 5 load and 1 store instruction. Since the pipeline needs to wait for the result of previous elements, it still takes 3 cycles to get an addition or multiplication result in the worst case. In "RAM phase", all data access is regular and the data is generally small, it does not cause any data transmission, so "RAM phase" time is 0.

By Formula (5), the worst GFLOPS is (3+4) * 2.7 / max((4*3),2.5) = 1.575 Gflops and actual measurement of 1.79 Gflops with out-of-order enable. Then we use ECM to model sn-sweep by Kerncraft or Intel Architecture Code Analyzer (IACA), it predict the GFLOPS is (3+4) * 2.7 / max((4*1),2.5) = 4.725 Gflops with the ideal instruction throughput. The reason is the data dependencies between instructions are stored in an index_i_up array which ECM fails to find and model it. Then from Table 4, we extend to predict performance with different data size from L1 to MEM by PRF and ECM model. The calculation of one point requires data transfer of five-eighths of the cache line. We can clearly observe that the prediction accuracy of the PRF is better the ECM when the amount of data is less than the L3 cache size. So we use the modeled time to apply the "Feedback optimization phase", and it can seen the performance cannot reach the scalar peak as the T1 takes extra time, so the model feed back pipeline is the primary bottleneck which is need to be optimized by Figure 4(a).

4.3.3 Optimization method and feedback

Table 4: GFLOPS performance in different data sizes for naive code, and comparison with PRF, ECM and measured.

	L1	L2	L3	MEM
T1 : add&mul	4*3	4*3	4*3	4*3
T2 : load&store	5/2	5/2	5/2	5/2
T3/T4/T5 : memory hierarchy	0/0/0	$\frac{5}{2}$*(2/0/0)	$\frac{5}{2}$*(2/4/0)	$\frac{5}{2}$*(2/4/10)
prediction GFLOPS by PRF	1.575	1.575	1.575	1.512
prediction GFLOPS by ECM	4.725	4.725	3.024	1.512
measured GFLOPS	1.79	1.773	1.821	1.623

By comparing the overhead of each phase, we can see that the computation instruction becomes a bottleneck, and the optimized GFLOPS can be achieved when the instruction pipeline is optimized: (3+4) * 2.7 / max((4*1),2.5) = 4.725 Gflops. We found that there is no data dependence between the diagonal elements and that the calculation order does not affect the final result by analyzing the DAG. As shown in Figure 11, we improved the algorithm to optimize the instruction pipeline by using the diagonal calculation order, and rearranged partial instruction with adding some calculation of subscripts, then the optimized GFLOPS is 4.43 Gflops. Based on this optimization, we can continue model sn-sweep, analyze bottlenecks by feedback optimization and increase SIMD operation to boost performance. Then we accelerate the addition and multiplication instructions by AVX2, and achieve a certain degree of vectorization and reach the optimal performance of 10.53 Gflops. Through our experiment, we finally convert a data dependence

problem into memory access instruction bottleneck. At the same time, the inaccurate model prediction will affect the developer to select the corresponding optimization method, resulting in the selected method does not have any optimization effect. This discovery also told us that the bottleneck will change under different optimization methods, so we need to model optimized kernel and explore more in-depth optimization. That is the meaning of feedback optimization.

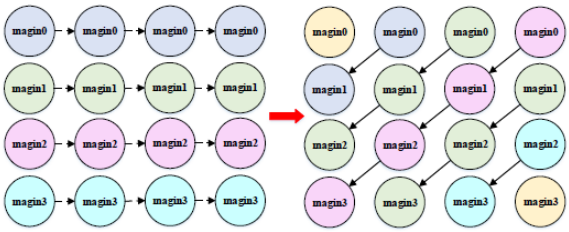

Figure 11: The computational order of eliminating data dependencies by feedback optimization

5 Conclusion

The PRF model described in this paper provided an insightful perspective to allow us to predict compute performance and to generate targeted optimization guidance. In this work, we introduced the PRF performance model, and described in detail about the "Process phase", "RAM phase" and "Feedback Optimization phase". Then we applied the model to convolution, SpMV and sn-sweep which are typical representatives of regular to irregular memory access and data dependence. It can be continue to cover more applications. In Table 5, comparison with Roofline model and ECM model, the proposed PRF model greatly improved predict accuracy for data dependence and irregular memory access by newly designed DAG and cache simulator, and finial achieved comprehensive feedback ability.

Table 5: Comparison with three performance models from different perspectives. The hook indicates the model has this function, the fork represents does not have and the bar represents incomplete function.

Method	Instruction count	Dependence analysis	Regular	Irregular	Feedback
Roofline	X	X	X	X	X
ECM	√	X	√	–	X
PRF	√	√	√	√	√

Reference

[1] FRIESE, Ryan D., et al. Generating performance models for irregular applications. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2017. p. 317-326.

[2] LI, Xin, et al. Statistical performance modeling and optimization. Foundations and Trends® in Electronic Design Automation, 2007, 1.4: 331-480.

[3] HÚDIK, Martin; HODOŇ, Michal. Modeling, optimization and performance prediction of parallel algorithms. In: 2014 IEEE Symposium on Computers and Communications (ISCC). IEEE, 2014. p. 1-7.

[4] DOMKE, Justin. Generic methods for optimization-based modeling. In: Artificial Intelligence and Statistics. 2012. p. 318-326.

[5] WILLIAMS, Samuel; WATERMAN, Andrew; PATTERSON, David. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2009.

[6] STENGEL, Holger, et al. Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model. In: Proceedings of the 29th ACM on International Conference on Supercomputing. ACM, 2015. p. 207-216.

[7] MALAS, Tareq M., et al. Multidimensional intratile parallelization for memory-starved stencil computations. ACM Transactions on Parallel Computing (TOPC), 2018, 4.3: 12.

[8] HAMMER, Julian, et al. Automatic loop kernel analysis and performance modeling with kerncraft. In: Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems. ACM, 2015. p. 4.

[9] HOFMANN, Johannes, et al. An analysis of core-and chip-level architectural features in four generations of Intel server processors. In: International Supercomputing Conference. Springer, Cham, 2017. p. 294-314.

[10] HOFMANN, Johannes, et al. Analysis of Intel's Haswell microarchitecture using the ECM model and microbenchmarks. In: International Conference on Architecture of Computing Systems. Springer, Cham, 2016. p. 210-222.

[11] HAMMER, Julian, et al. Kerncraft: a tool for analytic performance modeling of loop kernels. In: Tools for High Performance Computing 2016. Springer, Cham, 2017. p. 1-22.

[12] WEAVER, Vincent M.; TERPSTRA, Dan; MOORE, Shirley. Non-determinism and overcount on modern hardware performance counter implementations. In: 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, 2013. p. 215-224.

[13] WEAVER, Vincent M.; MCKEE, Sally A. Can hardware performance counters be trusted?. In: 2008 IEEE International Symposium on Workload Characterization. IEEE, 2008. p. 141-150.

[14] BHADAURIA, Major; WEAVER, Vincent M.; MCKEE, Sally A. PARSEC: hardware profiling of emerging workloads for CMP design. In: Proceedings of the 23rd international conference on Supercomputing. ACM, 2009. p. 509-510.

[15] TREIBIG, Jan; HAGER, Georg; WELLEIN, Gerhard. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In: 2010 39th International Conference on Parallel Processing Workshops. IEEE, 2010. p. 207-216.

[16] DAVIS, Timothy A.; HU, Yifan. The University of Florida sparse matrix collection. ACM Transactions on Mathematical Software (TOMS), 2011, 38.1: 1.

[17] ILIC, Aleksandar; PRATAS, Frederico; SOUSA, Leonel. Cache-aware Roofline model: Upgrading the loft. IEEE Computer Architecture Letters, 2014, 13.1: 21-24.

[18] KHATWAL, Ravi; JAIN, Manoj Kumar. Application specific cache simulation analysis for application specific instruction set processor. arXiv preprint arXiv:1406.5000, 2014.

[19] KANG, Hui; WONG, Jennifer L. vcsimx86: a cache simulation framework for x86 virtualization hosts. Stony Brook University.

[20] HAQUE, Mohammad Shihabul; PEDDERSEN, Jorgen; PARAMESWARAN, Sri. CIPARSim: Cache intersection property assisted rapid single-pass FIFO cache simulation technique. In: 2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, 2011. p. 126-133.

[21] DEY, Somdip; NAIR, Mamatha S. Design and Implementation of a Simple Cache Simulator in Java to Investigate MESI and MOESI Coherency Protocols. International Journal of Computer Applications, 2014, 87.11.

[22] FEIL, Manfred; UHL, Andreas. Wavelet packet decomposition and best basis selection on massively parallel SIMD arrays. In: Proceedings of the International Conference "Wavelets and Multiscale Methods"(IWC'98), Tangier. 1998.

[23] SHAHBAHRAMI, Asadollah; JUURLINK, Ben; VASSILIADIS, Stamatis. Performance comparison of SIMD implementations of the discrete wavelet transform. In: 2005 IEEE International Conference on Application-Specific

Systems, Architecture Processors (ASAP'05). IEEE, 2005. p. 393-398.

[24] BUTTARI, Alfredo, et al. Performance optimization and modeling of blocked sparse kernels. The International Journal of High Performance Computing Applications, 2007, 21.4: 467-484.

[25] LEE, Benjamin C., et al. Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. In: International Conference on Parallel Processing, 2004. ICPP 2004. IEEE, 2004. p. 169-176.

[26] PAUTZ, Shawn D. An algorithm for parallel Sn sweeps on unstructured meshes. Nuclear Science and Engineering, 2002, 140.2: 111-136.

[27] LIU, Jie, et al. Parallel Sn Sweep Scheduling Algorithm on Unstructured Grids for Multigroup Time-Dependent Particle Transport Equations. Nuclear Science and Engineering, 2016, 184.4: 527-536.

[28] SPRANGLE, Eric; CARMEAN, Doug. Increasing processor performance by implementing deeper pipelines. In: Proceedings 29th annual international symposium on computer architecture. IEEE, 2002. p. 25-34.

[29] GUIDE, Part. Intel® 64 and ia-32 architectures software developer's manual. Volume 3B: System programming Guide, Part, 2011, 2.

[30] INTEL, Intel. and IA-32 architectures software developer's manual. Volume 3A: System Programming Guide, Part, 64, 1.64: 64.

[31] KARAKASIS, Vasileios; GOUMAS, Georgios; KOZIRIS, Nectarios. Exploring the effect of block shapes on the performance of sparse kernels. In: 2009 IEEE International Symposium on Parallel & Distributed Processing. IEEE, 2009. p. 1-8.

[32] MUCCI, Philip J., et al. PAPI: A portable interface to hardware performance counters. In: Proceedings of the department of defense HPCMP users group conference. 1999.