Performance Tuning and Analysis for Stencil-Based Applications on POWER8 Processor

JINGHENG XU, HAOHUAN FU, WEN SHI, LIN GAN, and YUXUAN LI, Tsinghua University
WAYNE LUK, Imperial College London
GUANGWEN YANG, Tsinghua University

This article demonstrates an approach for combining general tuning techniques with the POWER8 hardware architecture through optimizing three representative stencil benchmarks. Two typical real-world applications, with kernels similar to those of the winning programs of the Gordon Bell Prize 2016 and 2017, are employed to illustrate algorithm modifications and a combination of hardware-oriented tuning strategies with the application algorithms. This work fills the gap between hardware capability and software performance of the POWER8 processor, and provides useful guidance for optimizing stencil-based scientific applications on POWER systems.

CCS Concepts: • **Computer systems organization → Multicore architectures**;

Additional Key Words and Phrases: POWER CPU, stencil, scientific applications, performance optimization, atmospheric simulation

This article is an extension of a conference paper: Evaluating the POWER8 Architecture through Optimizing Stencil-Based Algorithms published in ISPA 2016 (Xu et al. 2016a). We consider this work an improved edition of the conference paper with new contributions as follows. At the structural level, while the conference paper mainly focuses on performance optimization for typical stencil benchmarks on POWER8 platform, in this work, we further present a performance optimization framework that aims at facilitating the tuning efforts for stencil-based kernels and applications on POWER processors, and introduce a complex atmospheric simulation program as target to demonstrate the way of making proper algorithm modifications and fully combine the hardware-oriented capability and software features of the POWER system. At the method level, in this work, we further demonstrate some new optimization techniques (such as register-friendly data structure reconstruction and customized vector grouping strategy) and provide in-depth analysis. Such tuning methods are essential components of the systematic tuning guidelines of stencil-based applications based on the POWER system. Furthermore, a more sufficient technique explanation and more comprehensive related work are adopted in this article, thus, to show the existing efforts and challenges of performance tuning based on POWER systems, and to look deep into the insight of the architectural difference of POWER and x86 platforms.

L. Gan and J. Xu are supported by the National Natural Science Foundation of China (grant no. 61702297) and the China Postdoctoral Science Foundation (grant no. 2016M601031).
H. Fu and W. Shi are supported by the National Key Research & Development Plan of China (grant no. 2017YFA0604500), the National Natural Science Foundation of China (grant no. 91530323, 41661134014, 41504040, and 61361120098), and the Tsinghua University Initiative Scientific Research Program (no. 20131089356).
G. Yang and Y. Li are supported by the National Key Research & Development Plan of China (grant no. 2016YFA0602200). The support of the EU Horizon 2020 Research and Innovation Programme under grant agreement no. 671653 and the UK EPSRC (EP/L00058X/1, EP/L016796/1, EP/N031768/1, and EP/P010040/1) is gratefully acknowledged.
Authors' addresses: J. Xu, H. Fu, W. Shi, L. Gan (corresponding author), Y. Li, and G. Yang, Tsinghua University, Beijing, China (100084); emails: 18653236889@163.com, haohuan@tsinghua.edu.cn, shiwensmile@163.com, lingan@tsinghua.edu.cn, jflfy2255@163.com, ygw@tsinghua.edu.cn; All above authors are concurrently with the National Supercomputing Center in Wuxi, Wuxi, Jiangsu Prov. China (214000); W. Luk, Imperial College London, London, UK; email: w.luk@imperial.ac.uk.

1544-3566/2018/10-ART41
https://doi.org/10.1145/3264422

ACM Reference format:
Jingheng Xu, Haohuan Fu, Wen Shi, Lin Gan, Yuxuan Li, Wayne Luk, and Guangwen Yang. 2018. Performance Tuning and Analysis for Stencil-Based Applications on POWER8 Processor. *ACM Trans. Archit. Code Optim.* 15, 4, Article 41 (October 2018), 25 pages.
https://doi.org/10.1145/3264422

1 INTRODUCTION

The IBM POWER processors date back to the 1990s and provide a different approach of architectural design, compared to the x86 processor family. While the current supercomputers are mostly based on x86 processors, the recent OpenPOWER foundation has helped to promote the usage of POWER processors in various scientific computing domains, and some of the upcoming supercomputers targeting at 150PFlops (such as Summit (Laboratory 2017b) and Sierra (Laboratory 2017a)) will adopt POWER as the host processor.

Compared with the Intel and AMD x86 processors, the POWER processors demonstrate a number of different features, such as higher running frequency, preference for higher parallelism (both at the thread and the instruction level), larger cache sizes, and an extra level of L4 memory buffer, as well as customized support for encryption and decimal operations (Mericas et al. 2015).

These new and different features of the POWER processors not only provide the potential of achieving performance improvements for existing scientific computing applications, but also bring the challenge of identifying the right design approach and the appropriate tuning techniques for mapping the existing algorithms and kernels onto different POWER architectures.

Although there have already been earlier efforts (Friedrich et al. 2014; Adinetz et al. 2014; Ewart et al. 2015a; Stone et al. 2016; Ewart et al. 2015b) to investigate the POWER systems, most of them focus on the architectural features. Comprehensive discussions that systematically cover the tuning techniques and discuss the performance results are still seldom to be seen.

To resolve the above issue and to prepare the scientific kernels for POWER-based supercomputers, in this article, we explore the major hardware/software factors that determine the performance of stencils, a major consumer of compute cycles in scientific computing applications. By tuning and analyzing the performance of a set of different stencils, we try to identify the major architectural and tuning differences between POWER processors and the better-studied x86 processors, and to derive a general framework to guide the designing and tuning processes for POWER processors.

Our major contributions are as follows:

(1) Targeting the POWER8 processor, we perform a systematic tuning process to achieve almost optimum performances (46.7% of the processor peak) for 3D finite difference stencil benchmarks, thus to fill the gap between the hardware capability and the software performance on the POWER system;
(2) Based on these optimization guidelines and their evaluation, a performance tuning framework, which is the first framework aimed at guiding the tuning approach of stencil-based applications on recent POWER processors, is further provided to facilitate the performance tuning approach for programmers.
(3) We select two typical scientific applications (i.e., atmospheric simulation and seismic modeling) to demonstrate how to face the challenges when we optimize real-world applications on POWER processors. By analyzing the similarity and difference between benchmark optimization and performance tuning for real-world applications, we manage to

fully combine the hardware-oriented tuning techniques with the application algorithms, thus, to prove the effect of our tuning framework and optimization techniques.

As far as we know, this is the first work that presents systematic tuning guidelines and frameworks of stencil-based applications based on the recent POWER system, so as to facilitate the tuning efforts of POWER processors for stencil-based scientific applications. The corresponding optimization and analysis can be used as guidance for applications whose essential kernels largely rely on stencil-based algorithms.

2 RELATED WORK

Stencil-based algorithm refers to a class of iterative kernels that update array elements according to some fixed pattern (Gan et al. 2014a). As stencil computations are among the most important and frequently used kernels in modern scientific applications such as atmospheric simulation (Xue et al. 2014; Fu et al. 2016; Gan et al. 2016) and seismic modeling (Fu et al. 2014), stencil optimization based on mainstream processors has been a hot topic in recent years. To optimize the solving approach of stencil-based algorithms, stencil computation has been widely studied among different kinds of High Performance Computing (HPC) devices, such as Central Processing Unit (CPU), Graphics Processing Unit (GPU), Knights Landing (KNL), and Field-Programmable Gate Array (FPGA) (Datta et al. 2008; Kamil et al. 2010; Strzodka et al. 2011; Gan et al. 2013; Gan et al. 2014b; Gan et al. 2017; Cebrián et al. 2017). In 2010, in consideration of the increasing architectural Flop-to-Byte ratio of processors at that time, Nguyen el al. (2010) provided a novel 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs so as to release the memory access pressure. Such approach remarkably unleashed the performance potential of stencil-based programs on CPU and GPU processors at that time. In 2015, Jeffers et al. (2015) proposed a full-scale analyzation of optimization methodology for FD4-like Iso3DFD stencil kernel on Intel E5-2697 (v2) and Xeon Phi CO-7120P, and achieved 24.33% and 16.92% of their peak performance on E5-2697 and Xeon Phi 7120, respectively. In 2016 and 2017, Chao et al. and Haohuan et al. optimized the stencil-based applications on the world's fastest supercomputer Sunway-TaihuLight (Yang et al. 2016; Fu et al. 2017a), and their achievements won the Gordon Bell Prize of these two years, respectively.

As some of the world's fastest supercomputers targeting at 150PFlops (such as Summit (Laboratory 2017b) and Sierra (Laboratory 2017a)) would adopt POWER9 as the host processor, the next generation of POWER CPUs will play a huge role in high performance computing once again. However, the design of POWER systems has obvious differences compared with that of other contemporaneous multi-core processors. First of all, benefiting from the higher instruction throughput, POWER processors could issue and commit more instructions per cycle. As a trade-off, the vector units are not as wide as other contemporaneous CPUs (Reguly et al. 2015). Secondly, POWER processor enables a higher number of simultaneous multi-threading (SMT) per core. While such design increases the number of executed instructions per cycle for some multi-bottleneck applications, the extra overhead of thread management and hardware resource competition might decrease the overall performance (Xu et al. 2016a). Thirdly, while the bandwidth of POWER processors is remarkable higher than that of the contemporaneous Intel CPUs, the memory latency becomes the new problem in the performance tuning approach (Reguly et al. 2015). Due to these new features, while existing efforts of stencil optimizations based on mainstream platforms such as CPU and GPU have resulted in good performance (Johnsen et al. 2013), some of the classical stencil-based benchmark and applications are still suffering the low-computing efficiency on the POWER8 platform. (As demonstrated in Table 1, only similar performance efficiency (the ratio of achieved performance over the theoretical peak) is achieved on POWER8 compared with Intel CPUs. However, due to the large fast-memory buffers and other features of POWER

Table 1. Application Optimization Efforts on POWER8 Platforms

Program	Inclusion	**Effect** (percentage of peak)	Evaluation
Lattice Boltzmann [Adinetz et al. 2014]	OpenMP, SMT mode Cache blocking	POWER8: 29% Intel X5680: 70%	Performance is not as good as expected
Airfoil [Reguly et al. 2015]	OpenMP, SMT mode SIMD, NUMA	POWER8: avg. ~7% Intel E5-2650(v3): avg. ~6%	Performance is not as good as expected
Black-Scholes [Reguly et al. 2016]	OpenMP, SMT mode SIMD, NUMA	POWER8: 9.5% – 19% Intel E5-2650(v3): 15% – 25%	Performance is not as good as expected
SpMV [Liu et al. 2016]	OpenMP, SMT mode Customized methods	POWER8: 0.72% – 6.73% Intel: Unmentioned	Unrelated to stencil-based applications
There are also some other works targeted to optimize various kinds of applications on the POWER8 platform (also mentioned in this section), but their effects are not able to be calculated according to the information provided by the authors. So we do not list them in this table.			

system, we expect to achieve a 1.5x performance efficiency on stencil-based applications compared with contemporary Intel CPUs.) Exploring the performance potential of stencil-based algorithms on POWER platforms has thereby been an urgent demand.

With the rapid development of OpenPOWER all over the world, as well as a part of large-scale future supercomputers would consist of POWER processors, exploring the performance potential on POWER architecture started to be a hot topic in recent years. The first article related to POWER8 was published on International Conference on IC Design and Technology (ICICDT) (Friedrich et al. 2014) in the middle of 2014. The paper, which demonstrates a couple of hardware innovations of the new computational chip, is mainly focusing on the hardware design and I/O capabilities of POWER8. Some other works (such as Fluhr et al. (2014) and Sinharoy et al. (2015)) were published then as a complement.

After the hardware description, the performance evaluation of micro-benchmarks and real-world applications of POWER8 has been published since the end of 2014 (Mericas et al. 2015; Adinetz et al. 2014; Ewart et al. 2015a; Liu et al. 2016). For instance, in 2014, Adinetz et al. (2014) provided a set of hardware-feature evaluations, such as operation throughput/latency and memory latency/bandwidth, by using standard testing micro-benchmarks such as Stream. After this, the author adopted three scientific applications to evaluate their performance on POWER8. Such approach is the first work to explore the software performance on POWER8; however, performance tuning techniques are not fully involved in this work. A similar approach could also be found in Berreth et al. (2016) and Ashworth et al. (2016). In 2016, based on the testing results of micro-benchmarks, Reguly et al. (2016) proposed the system-level analysis of POWER8 and provided a set of performance tuning techniques (such as OpenMP, SMT, and non-uniform memory access (NUMA) control) for a set of stencil-based applications with different kinds of bottlenecks. This work indicates that the POWER8 processor is capable of delivering high performance for a wide range of kernels and applications. However, though a remarkable performance benefit is able to be achieved in this work, only some general tuning techniques are involved, and the author does not provide necessary modification to application algorithms to further increase their performance efficiency.

To fully fill the gap between the hardware capability and the software performance on POWER system, besides providing a set of tuning techniques and their analysis to three representative stencil benchmarks, we also need to introduce some application-level programs as target so as to demonstrate how to face the challenges when we optimize complicated cases. Nowadays, atmospheric simulation and seismic modeling programs are undoubtedly two important representatives in HPC studies. From 2015 to 2017, five (among 13 in 3 years) of the ACM Gordon Bell Prize finalists (Fu et al. 2017a; Fu et al. 2017b; Yang et al. 2016; Ichimura et al. 2015; Rudi et al. 2015)

Table 2. Main Hardware Parameters of IBM POWER8 and Contemporaneous Intel CPU

	Peak Perf & Frequency	Physical Cores & Max Threads	Pipeline & SIMD Units per Core	On-chip Memory (Per Core)	Off-chip Memory (Per Chip)
IBM POWER8 (Aug 25, 2013)	638GFlops (double, 2 chips) 3.325GHz (Base)	12-cores / chip; 96 threads in maximum	16 pipelines; 64 * 128-bit Vector-Scalar Registers (VSRs)	L1-d_cache: 64KB * 12 L2-cache: 512KB * 12 L3-cache: 8096KB * 12	L4-cache: 128MB Main Mem: 1TB
Intel E5-2697 (Sep 10, 2013)	518GFlops (double, 2 chips) 2.7GHz (Base)	12-cores / chip; 24 threads in maximum	14 pipelines; 16 * 256-bit Vector Registers	L1-d_cache: 32KB * 12 L2-cache: 256KB * 12	L3-cache: 30MB Main Mem: 768GB

are related to these two fields, and two of them finally won the Gordon Bell Prize (Yang et al. 2016; Fu et al. 2017a). Thus, in this work, we adopt a seismic modeling program (Reverse Time Migration (RTM)) as well as an atmospheric simulation algorithm (dynamic part based on shallow water equations (SWE)) as target programs to demonstrate and evaluate the effectiveness of our optimization techniques and tuning framework.

3 BACKGROUND

3.1 Brief Introduction to POWER8 Processor

As the typical version of the POWER series, the IBM POWER8 processor is designed for both high thread-level performance and system throughput on a variety of workloads. On each POWER8 chip, there are up to 12 processor cores, each of which can be clocked at up to 4.2GHz and is capable of running up to eight threads per core in SMT mode. Instruction level parallelism is also exploited with increased dispatch and execution bandwidth. In a certain cycle, each of the POWER8 processor cores could fetch, decode, and dispatch up to 8 instructions, and issue and execute up to 10 instructions (Hall et al. 2014).

The main hardware parameters of IBM POWER8 and contemporaneous Intel E5-2697 CPU are demonstrated in Table 2 (data source: (IBM, Wikipedia, and Anandtech IBM et al.), (Intel, Wikipedia, and stuffedcow Intel et al.)). From the table and relative papers, we could find out that the POWER8 processor core enables deeper out-of-order execution, better branch prediction, larger cache spaces, and wider bandwidth compared with contemporaneous Intel processors. In return, narrower Single Instruction Multiple Data (SIMD) units and less load buffers are equipped on POWER8, and the single-thread performance of POWER within one core is not as good as Intel, due to the lack of units such as loop stream detectors. Therefore, when exploring the optimal performance of programs on POWER, we should propose a set of novel optimization methods or readjust the existing tuning techniques to let them fit into the POWER processor.

3.2 Evaluation of IBM S-824L POWER System

IBM S-824L is the first server to leverage OpenPOWER Foundation technology to dramatically accelerate modern programs such as technical computing applications. The hardware includes two 12-core POWER8 processors whose core frequency is set to 3.325GHz throughout this study. In addition, the POWER8 system has been configured to use a little Endian OS distribution (Ubuntu 14.10).

For a certain processor, the architectural feature of hardware identifies its peak performance and memory bandwidth. The achievable rate values can be obtained by executing standard micro-benchmarks such as LINPACK (Dongarra et al. 2003) and Stream (McCalpin 1995). As

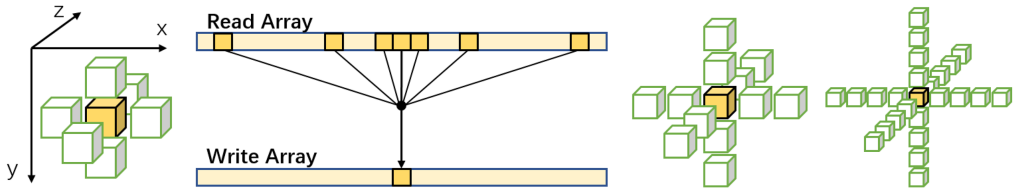


Fig. 1. Stencil visualization. 7-point Jacobi stencil in 3D space; Mapping of stencil from 3D space onto 1D array; 13-point FD4 stencil in 3D space; 25-point FD8 stencil in 3D space.

the theoretical peak performance and bandwidth of IBM S-824L remains 638.4 GFlops (double precision) and 460.8GB/s, respectively (Xu et al. 2016a), achieved values on Linpack and Stream Triad benchmarks give us an upper bound performance and bandwidth for the S-824L platform: 590GFlops in double precision and 320GB/s. The detailed testing methodology could be found in Ewart et al. (2015a) and Jeffers and Reinders (2015). However, the calculation of peak performance indicated above includes some strong assumptions that are hard to achieve in real-world programs. First of all, the number of add and multiply should be exactly the same so that the fused Multiply-Add instruction could be fully used. Secondly, the device should be equipped with caches that have an extremely high bandwidth and low latency so that the cache access would never block the program execution. This defines a type of perfect memory subsystem where once data in memory is loaded into the cache, it is instantly available without stalling the calculation.

3.3 Stencil Benchmarks

Our first kernel is the basic 7-point 3D Jacobi stencil, which is frequently used as a benchmark for stencil study (Datta et al. 2008) and widely applied in real-world applications such as POP (Smith et al. 2010) and GRAPES (Browne et al. 2000). The shape of Jacobi is shown in Figure 1. To update each grid point, seven input variables as well as one output variable need to be accessed, with six additions and two multiplications being performed at the same time, as demonstrated in Algorithm 1. From Figure 1, it is not hard to predicate that without careful design, cache misses caused by the remote access of the array elements would stall the program to a great extent.

ALGORITHM 1: Demonstration of the Jacobi Algorithm

```
1: void stencil_computation(double dst[], double src[]) {
2: for z ← 1 to z_n + 1
3:     for y ← 1 to y_n + 1
4:         for x ← 1 to x_n + 1
5:             dst[z,y,x] = c1*src[z,y,x] + c2*(src[z+1,y,x]+src[z,y+1,x]+
                   src[z,y,x+1]+src[z,y,x-1]+src[z,y-1,x]+src[z-1,y,x]);
6: }
```

The 4th-order and 8th-order finite-difference stencils (FD4 and FD8, as shown in Figure 1) are also two of the most widely used stencil kernels. Though the Flop-to-Byte ratio is approximately the same as Jacobi, long tiles in all three dimensions would definitely lead to discrete accesses to the main memory, vitiating the cache data reuse mechanism provided by modern micro-processors.

To provide a full-scale analysis of stencil optimization techniques, in this work, we set the grid size of all three stencils as 960*640*1280, which is larger than the on-chip cache capacity of mainstream processors, including POWER8. Due to the memory access pattern of FD stencils, only one direction (x-axis in our case) has the data consecutively stored in memory. Thus, access to memory for other directions is very expensive. In addition, due to the simple computational operations for

Table 3. Left: Critical Compilation Flags for xl and gcc Compilers; Right: Corresponding Single-core Performance Result of gcc and xl Compilers (GFlops, in Double Precision)

	xl compiler	gcc compiler
POWER8	-qarch=pwr8 -qtune=pwr8	-mcpu=power8 -mtune=power8
General	-qhot –O3 –q64	-O3 –m64
OpenMP	-qsmp=omp	-fopenmp -fopenmp-simd
Vectorization	-qsimd=auto	-mvsx –maltivec
Prefetching	-qprefetch=aggressive	
Permutation	-qaltive=be	
Others	-qalias=noansi -qdebug=npwr7ra	-mveclibabi=mass -lmass_simdp8 -ftree-vectorize

	xl compiler	gcc compiler	speedup (xl to gcc)
Jacobi	4.35	3.17	37.2%
FD4	4.57	4.03	13.4%
FD8	4.27	2.37	80.2%

each mesh element of FD stencils, reuse of cache data is strictly limited, which further enhances the memory access pressure.

3.4 Challenges

Aiming at providing a set of optimization guidelines so as to facilitate the performance tuning approach for a stencil-based algorithm on POWER processors, there are multiple challenges we need to face. First of all, due to the architectural difference between POWER processors and other contemporaneous multi-core processors (details can be found in Section 2 regarding a wider pipeline but narrower vector units, a higher SMT number per core, and a higher bandwidth but a longer latency), the existing tuning methods may not work on POWER systems. Thus, we should propose a set of novel optimization methods or readjust the existing tuning techniques to let them fit into the POWER processor. Secondly, to clarify the puzzling problem during the tuning approach and prove the effectiveness of our optimization techniques, we should provide detailed analysis and full-scale evaluations for the achieved performance. Thirdly, performance tuning of benchmarks and real-world applications are totally different. Based on the tuning guidelines of stencil benchmarks, we should also choose applications to demonstrate how to properly adopt the optimization techniques in real-world cases.

4 OPTIMIZATION TECHNIQUES TOWARD STENCIL BENCHMARKS

In this section, we introduce major optimizing techniques based on the POWER architecture. To measure the impact of these tuning techniques, preliminary results and analysis are proposed right after the introduction of each optimization method. The thorough experimental results and analysis would be demonstrated in Section 5.

4.1 Lightweight Tuning and Loop Interchange

This step indicates how to choose the best compiler options based on the POWER8 processor. With this study, specific focus has been given to the evaluations and optimizations of the C/C++ set as the stencil-based scientific applications are implemented adopting these languages (Ewart et al. 2015a) (Adinetz et al. 2014). Although this step is simple and straightforward, it often leads to significant performance improvements and should be considered as the starting point for performance tuning.

The IBM XL C/C++ compilers, which could be deployed on POWER, BlueGene/Q, and z Systems hardware architectures, are applied to transform C or C++ source code to fully exploit IBM hardware (IBM Europe and Announcement 2014). An evaluation of the two provided compilers (gcc and xlc) is presented along with critical compiler options (Table 3). We should notice that most of the compiler optimizations such as data prefetching would be adopted automatically after simply employing these compiler options. According to the experimental results shown in Figure 3, a remarkable performance benefit could be obtained by simply adopting these options. Besides,

Table 4. SMT Testing of Scalar and Vector Program (24-core, Double Precision)

Unit: GFlops	Scalar Version				Vector Version			
	SMT=1	SMT=2	SMT=4	SMT=8	SMT=1	SMT=2	SMT=4	SMT=8
Jacobi	58.43	58.92	43.44	26.81	101.66	100.01	20.47	10.63
FD4	75.25	41.31	20.74	11.52	178.24	77.01	26.09	12.63
FD8	71.24	44.95	50.09	56.93	160.52	128.05	74.36	23.07

compared with gcc compiler, an averaged 40% speedup is able to be achieved for three stencil benchmarks by employing the XL compiler, as demonstrated in the right part of Table 3.

Loop interchange is the process of exchanging the order of two iteration variables used by a nested loop. We usually perform loop interchange to ensure that the elements of a multi-dimensional array are accessed in the same order as they are stored in memory, so as to improve the data locality and cache hit rate. However, nowadays most programmers would pay attention to this when they write their program. To provide a fair comparison, the algorithms employed in this work adopt the most efficient loop structure naturally.

4.2 Combination of OpenMP and Simultaneous Multi-threading

The mapping of threads to multiple CPU cores is a critical factor for the performance of modern concurrent applications beyond a modest number of threads. The XL compilers provide a full implementation of the OpenMP specification, and, thus, programmers could use OpenMP runtime control variables to arrange the thread layout. Specially, in S-824L, a total of 24 threads is required to occupy all physical cores in single thread mode (ST mode). Further performance gains may be achieved by employing multiple threads on one physical core in SMT mode. With SMT, each POWER8 core could present up to eight hardware threads and issue more instructions per cycle by filling the void execution pipelines of processor cores. However, the overheads of threads management and hardware resource competition will grow along with the increase of thread numbers. And among these two influences, while the overheads of thread management (such as context switch) is relatively stable and has been discussed in some previous publications (such as Adinetz et al. (2014) and Liu et al. (2016)), the impact caused by hardware resource competition (such as cache space, register files, issue queue entries, etc.) would change significantly according to different applications.

Though the optimal SMT thread number (which could be 1, 2, 4, or 8) for different applications is various and needs to be explored by experiments, for a fixed SMT thread number, the best result would generally be achieved by employing a round-robin allocation with stride $s = min(Num_{SMT}, \frac{Num_{core} * Num_{SMT}}{Num_{threads}})$ to take full usage of hardware resources (which corresponds to the results demonstrated in Waldspurger (1996)). Namely, if SMT=8, the optimal stride number should be $s = min(8, \frac{192}{Num_{threads}})$ in a POWER8 server with 24 processor cores.

Due to the low Flop-to-Byte ratio of stencil-based algorithms, the memory access units of each processor are always kept busy. As a result, the benefits achieved by adopting simultaneous multi-threading modes could be even lower than its overhead. Therefore, SMT=1 (ST mode) or SMT=2 are usually the optimal choice in simple stencil-based algorithms such as FD stencils demonstrated in this part. In our tuning approach of three kinds of stencils, a simple code generator employing *sed* was written to find out the best choice of SMT mode. Experimental results demonstrated in Table 4 approve our judgment indicated above. However, we should notice that when it comes to complex stencil algorithms, the optimal SMT choice may change to four or eight. We will discuss this part in detail in Section 6.

4.3 Non-Uniform Memory Access (NUMA) Aware Allocation

As indicated in Section 3.4, compared with the tuning approach of other contemporaneous Intel processors, while stencil-based applications benefit from the wider bandwidth and larger cache capability on POWER processors, the longer latency may become the performance-killer. Thus, careful NUMA-aware design is of vital importance in POWER processors so as to make sure that each processor is accessing its own local memory rather than accessing the non-local memory.

In stencil computations, the source and destination grids are each individually allocated as one large array. For each point of the array, before calculation, it would be initialized at first. This initializing approach would deterministically specify the thread that updates each mesh point. Without a careful NUMA-aware design, long-latency remote memory access is highly probable to occur, resulting in a performance hit of the whole program. To avoid this non-local memory access, we specify a suitable affinity configuration for the OpenMP threads by setting suitable OpenMP primitives and employing *numactl* instruction to associate a physical memory address with a given logical address. As a result, a "first touch" page-mapping policy would be implemented and data is correctly pinned to the physical processor to initialize it. Figure 3 demonstrates the results for the same computation with and without NUMA mapping. In this experiment, the compact affinity mode could improve the performance by 50% at most over the default mode.

4.4 SIMD Vectorization and Register Blocking

As the POWER8 processor adopts 128-bit vector registers rather than 256-bit or even wider vector units on other mainstream processors, how to balance the usage of general-used registers and vector-scalar registers becomes a challenging task. Therefore, we should pay special attention to properly employ SIMD on POWER8 processor.

In POWER8 architecture, the introduction of Vector Scalar eXtension (VSX) increases the parallelism by providing SIMD execution functionality for floating point computing to improve the performance of the HPC applications. To increase the opportunities for vectorization, a unified register file (a set of Vector-Scalar Registers (VSR)) supporting both scalar and vector operations is provided to eliminate the impact of vector-scalar data transfer in memory. This feature allows us to simultaneously implement vectorization, loop unrolling, and register blocking.

Take the double precision FD4 stencil for example; vector-scalar vector registers enable us to load and compute two points of the data space simultaneously. During the implementation of vectorization approach, there are two key components that can affect the overall performance: (1) 32 general-purpose registers (GPRs) used for temporary storage and register renaming and (2) 64 128-bit VSRs to store vector data. Due to the algorithm features (a huge amount of register renaming and temporary storage) of stencil-based kernels, the GPRs would be used up first. Thus, there are two optimizing strategies including the GPR-oriented programming, which would stop as far as the program has run out of GPRs; and the VSR-oriented programming, which would vectorize more mesh elements until the VSRs are used up.

According to the experimental results demonstrated in Table 5, it is hard to say which strategy is better to stencil-based kernels after the SIMD step (Step 3). However, the performance potential of GPR-oriented programming is apparently better than that of the VSR-oriented strategy after the cache blocking step (Step 4).

4.5 Cache Blocking

Considering the larger cache capability in POWER processors, how to take full advantage of the fast on-chip memory is of vital importance in the program tuning approach. To avoid the cache miss of stencil solvers, we first decompose the entire grid into a chunk of blocks whose sizes are

Table 5. Performance of Two Kinds of Vectorization Strategies

24-core, SMT=1, Unit: GFlops		Jacobi	FD4	FD8
Step3: SIMD	GPR-oriented	59.81	74.12	82.51
	VSR-oriented	60.28	68.42	83.28
Step4: Cache Blocking	GPR-oriented	101.66	178.24	160.41
	VSR-oriented	96.98	150.36	150.84

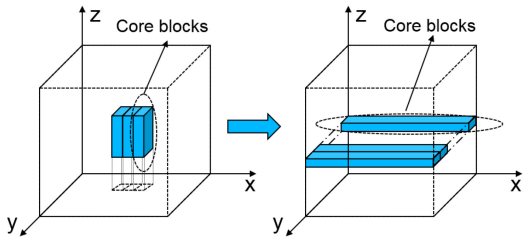


Fig. 2. Left: 3D Decomposition methodology. Right: 2D Decomposition methodology without cutting *x*-axis.

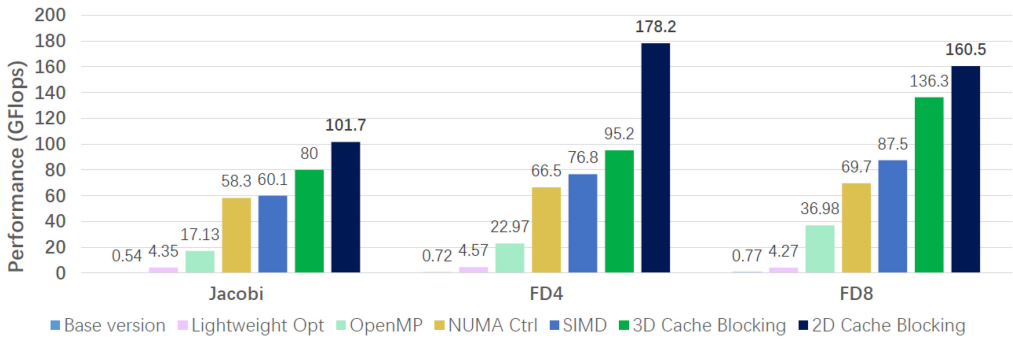


Fig. 3. Optimized performance results for three stencils (Unit: GFlops, in double precision). The compiler version and OpenMP version are XLC 15.1.3 and OpenMP 3.1, respectively.

exactly the same to the L3 cache of POWER8 (8MB). L3 cache miss rate is consequently reduced, resulting in a speedup of more than 20% in general.

While the above step focuses on minimizing the intra-block L3 cache miss, we could further reduce the inter-block L3 cache miss by keeping the *x*-axis (the least unit stride) undivided, as demonstrated in the right of Figure 2. Comparing with the 3D cache blocking methodology shown in the left, more data in the *y*-axis could be reused in the 2D decomposition, and, in addition, the program could take full advantage of the prefetching mechanism (Rivera and Tseng 2000). These two features could further exploit the locality of the L3 cache and result in a more than 50% speedup as shown in Figure 3.

5 PERFORMANCE AND ANALYSIS

5.1 Performance Analysis

Since scientific computing relies primarily on double precision, in this section, a set of experiments is presented to evaluate the effectiveness of optimization techniques based on three FD stencil kernels in double precision. The performance results of the base version are achieved by

Table 6. Performance Speedup of Stencil Benchmarks by Adopting Different Optimization Methods

	Base Version	Lightweight Opt	OpenMP + NUMA Control	SIMD	Cache Blocking	
					3D	2D
Jacobi	1x	8.05x	13.4x	1.03x	1.33x	1.69x
FD4	1x	6.35x	14.6x	1.16x	1.24x	2.32x
FD8	1x	5.54x	16.3x	1.26x	1.56x	1.83x

simply running the benchmark programs (using gcc compiler) without adopting any compiler options. According to the experimental results demonstrated in Figure 3 and Table 6, though the lightweight optimization methods are simple and straightforward (as introduced in Section 4), significant performance benefits could be achieved by the appropriate usage of these tuning techniques.

When we scale the program from 1 core to 24 cores (from Lightweight Opt to NUMA Ctrl), the speedup of three stencil kernels is 13.4x, 14.6x, and 16.3x, respectively, which is a little far from the linear scalability. These results demonstrate that all three stencil kernels are memory-bound programs when running on the POWER8 processor. The rising speedup from Jacobi (FD2) to FD8 reflects the gradual increase of arithmetic intensity of these three kernels. Within these tuning techniques, the optimal combination of SMT and OpenMP is various among different algorithms; how to select the best choice is an important task for programmers to face. In addition, the effect of NUMA-aware design on POWER is more significant than that of Intel CPUs (Jeffers and Reinders 2015), which is accordant with the hardware features (deeper pipeline, larger cache space, but longer latency) of POWER8.

The effects of vectorization could be obtained when we double the instruction-level parallelism of three kernels. Since the Jacobi stencil is bounded by main memory accesses even in the most arithmetic-intensive part of the kernel, only 3% speedup is able to be achieved by applying vectorization. As a comparison, due to the huge amount of data reuse in FD8 computation, its memory access pressure is not as large as Jacobi. Therefore, a speedup of more than 25% could be obtained for the FD8 stencil. As expected, the speedup of the FD4 stencil kernel (16%) is right between the speedup of Jacobi (3%) and FD8 (26%), which approves our judgement mentioned above.

As mentioned in Section 4, cache blocking reduces the L3 cache misses on POWER8 processors, which could result in a remarkable performance boost for memory bound programs, such as the three FD stencil benchmarks applied in this article. Thus, a speedup of up to 2.32x is able to be achieved by properly adopting the cache blocking. Combining all these tuning techniques together, a significant speedup could be achieved in all of the three stencil kernels, as demonstrated in Figure 3 and Table 6.

5.2 Performance Evaluation

Since the peak performance of different hardware platforms is various, to provide a full-scale evaluation and fair comparison, in this part, we apply the hardware usage efficiency, which is calculated by the measured performance dividing its peak performance, as the performance metric, thus, to compare the benchmark performance in POWER8 to that of other mainstream platforms.

Tables 7 and 8 indicate the performance of stencil benchmarks on POWER8 and other mainstream platforms, respectively. From the experimental results, we could figure out that the performance efficiency (the ratio of achieved performance over the theoretical peak) of similar stencil programs on POWER8 is more than two times higher than that of other mainstream platforms.

Table 7. Experimental Results of POWER8

POWER8, 24-core	Standard Performance	Non-standard Performance[1]
Jacobi	17% (102GFlops)	25.5% (153GFlops)
FD4	29.2% (175GFlops)	46.7% (280GFlops)
FD8	26.7% (160GFlops)	46.1% (276GFlops)

[1]Non-standard Performance: Applying a trick in stencil computations to enhance the arithmetic intensity and balance the addition and multiplication. For instance, change the $c_2*(src_1+src_2+...+src_n)$ shown in Algorithm 1 to $c_2*src_1+c_2*src_2+...+c_2*src_n$.

Table 8. Experimental Results of Other Platforms

Platform	Core Number	Stencil Type	Performance	Standard or not
Xeon E5355 [Datta et al. 2008]	4	Jacobi	6.1%	Standard
Xeon E5645 [Gan et al. 2014]	16	Jacobi	4.9%	Standard
Xeon E5645 [Gan et al. 2014]	16	FD6	10.2%	Standard
QS22 PowerXCell [Datta et al. 2008]	16	Jacobi	7.5%	Standard
E5-2697 v2 [Jeffers et al. 2015]	24	FD4-like	24.3%	Non-standard

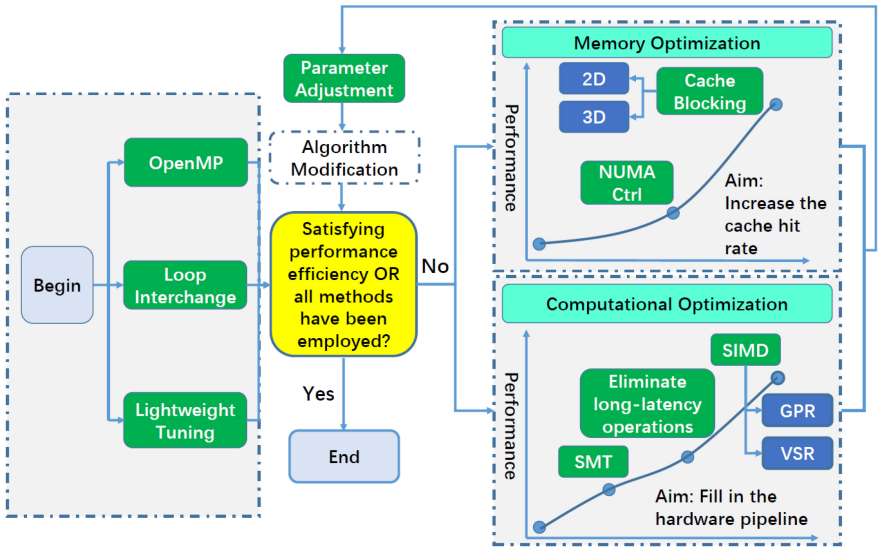


Fig. 4. Stencil-based application tuning framework on POWER system. Different kinds of optimization methods should be adopted according to the bottleneck of the target program.

(To provide fair and convincing comparison results, we directly adopt the optimization results achieved by the expert of corresponding platforms.) These results further prove the efficiency of our tuning guidelines as well as the computational ability of POWER processors.

Specifically, we would like to mention that in most publications, the standard performance (explained in the bottom of the Table) is applied as the performance metric to describe the experimental result of stencilbased applications. However, the non-standard performance is being used sometimes. To provide a full-scale evaluation of our optimization results, the standard performance metric is employed in most parts, and the non-standard performance is also mentioned in Table 7.

5.3 Performance Tuning Framework toward Stencil-Based Kernels

Based on the optimization techniques and analysis indicated above, we could further derive the performance tuning framework of stencil-based kernels based on the POWER8 platform. As shown in Figure 4, the framework contains three major modules and a judge box, and is suitable for optimizing stencil-based algorithms.

When a stencil-based kernel comes into the optimization framework, the first three tuning techniques that should be adopted are OpenMP, loop interchange, and lightweight tuning. While OpenMP could enable the basic thread-level parallelization of processors, the adoption of loop interchange and lightweight tuning (namely choose the optimal compiler as well as compiler options) would ensure a cache-friendly data structure and activate the automatic optimizations of compilers.

After adopting the optimization methods mentioned above, programmers should judge if the performance efficiency of our target program has met their requirements. If so, we could end our optimization. Otherwise, we need to continue the optimization. If the optimization task continues and the program is mainly bounded by bandwidth or throughput, we should try our best to increase the cache hit rate and decrease the memory access latency. Specifically, cache blocking and NUMA-Aware allocation should be adopted in such case. Otherwise, if the program is computation bound, then tuning methods such as SMT, SIMD, and eliminating complex operations should be taken so as to fill in hardware pipelines. The detailed information of how to properly employ these tuning techniques is indicated in Section 4.

After these steps, we need to readjust the parameters (such as compiler options, SMT value) if needed. Besides the general optimization methods indicated above, for some extremely complex programs, we have to adjust the data structure or even the algorithm so as to fully combine the algorithm features with the hardware characters. We come to this part in detail in Section 6.

As we have made sufficient explanations of how to optimize stencil-based benchmarks on POWER8 in a previous part, in the following sections, we will choose two most typical scientific applications (i.e., atmospheric simulation and seismic modeling) to show both the similarities and differences between benchmark optimization and performance tuning for real-world applications, so as to demonstrate how to combine the hardware-oriented tuning strategies with the application algorithms.

6 APPLICATION I: ATMOSPHERIC MODELING

6.1 Algorithm and Challenges

As demonstrated in Section 2, in this section, we choose the global shallow water equations (SWEs) based atmospheric solver, which is a similar atmospheric modeling program to the winner program of the Gordon Bell Prize in 2016 (Yang et al. 2016), as the first study case of this work. The global SWEs, which constitute a simplified atmospheric-like fluid prediction model that is able to exhibit the essential features of the global atmosphere, are one of the most popular equation sets that are selected to simulate the global climate behavior.

Algorithm 2 demonstrates the algorithm of the SWE solver (Yang et al. 2013; Xu et al. 2016b), the kernel of which is a 13-point SWEs stencil. The solving approach of the SWEs algorithm is basically a stencil computation involving 13 cells in a neighboring domain, as demonstrated in Figure 5.

Besides the challenges in the tuning approach of stencil benchmarks, we have to face some new challenges when we optimize real-world applications such as the SWE solver: (1) Long latency operations such as sqrt() and pow() decrease the number of instructions that can be finished within one cycle. (2) Instruction dependency and branches are unavoidable in SWE solvers, which would result in both irregular memory access and imbalanced workload distribution among different threads. (3) Lacking of parallelization-friendly data structures makes it hard to employ some parallel characteristics (such as VSX instruction set to trigger SIMD) of POWER8.

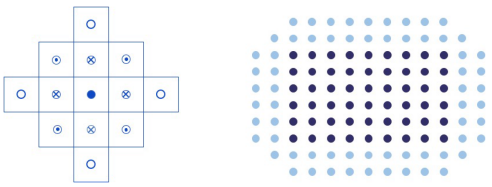


Fig. 5. **Left**: 13-point stencil used in SWE, where the adjacent 13 elements are needed to calculate the centric one. **Right**: Solid dots and halo elements involved in an SWE grid.

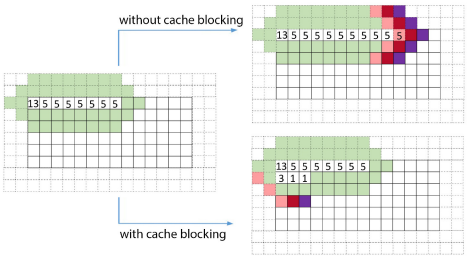


Fig. 6. Data locality without and with cache blocking. In an ideal case, by adopting the cache blocking strategy properly, if one element is driven out of the cache, it would never be used again.

ALGORITHM 2: Algorithm of SWE Solver

```
1: CPU Begin
2:     Data Initialization for CPU(input_x,input_hs, and so on)
3:     for (j, i) ← (0, 0) to (nyl, nxl) do the SWEs Stencils
4:         Calculate Coordinate
5:         Calculate Fluxes{
6:             State Reconstruction
7:             Riemann Solver}
8:         Compute Source Terms
9:     end for
10: CPU End
```

6.2 First-Round Optimization of the SWE Solver

In this part, we start to apply the tuning methods within the framework (Figure 4) so as to provide a first-round performance tuning of the SWE solver. According to the tuning framework, three basic optimization techniques are adopted at first. We adopt 24-threads OpenMP in single-thread mode (ST mode) and employ xl compiler along with lightweight tuning techniques indicated in Section 4. As indicated above, the loop interchange is automatically adopted in this application so no more additional modification needs to be done to employ this method. Then, we come to the first judgment box and it turns out that the application is memory bound and further optimizations need to be adopted.

To deal with memory bound programs, NUMA control and cache blocking should be adopted according to the tuning framework. The way of adopting NUMA-Aware allocation is almost the same as what we described in Section 4.3, and we will not go into detail here. As for cache blocking, to simplify the description, we only discuss one matrix adopted in the SWE solver. Suppose the size of the matrix is N*N, and L1-Cache becomes full after element (i, j) is processed. If we continue to process elements on the same row, we need to load five more new elements from the main memory every time when we process a mesh element. To make things worse, when we finish processing the first row and get to the first element of the second row, there is no cache-data available for reuse, since the elements once loaded in cache have been ejected already. In this case, 13 elements have to be loaded when a new line is processed; thus, the number of elements we need to load to finish processing the whole matrix should be $n1 = 5 * N^2 + 8 * N$, as demonstrated in the upper part of Figure 6. While, if we adopt the strategy shown in the below part of Figure 6 and turn to dealing with the first element of the second row after the element (i, j) is processed, only three

new elements are loaded in the next step and only one new element will be needed to process a new mesh cell in the future steps. In this case, for each i*N block, (i+2)(N+4) times data fetching is required, so for the N*N matrix, only $n2 = (1 + \frac{2}{i}) * N^2 + (4 + \frac{8}{i}) * N$ times global memory access is required in total, which is far less than the previous case. (To simplify the discussion, we do not involve cache line, but we should notice that the conclusion remains unchanged even in consideration of cache line.)

So far, we manage to apply the three basic tuning methods as well as memory optimization techniques shown in the framework. However, even if we choose the best SMT choice at this moment, the performance efficiency is 12.8% (75.35 / 590), which is not very satisfying. Thus, another round of optimization is needed and computational-oriented tuning techniques should be employed.

As indicated above, there are several long-latency operations such as TAN and *pow*() involved in the computation of the state reconstruction step. Such operations would heavily degrade the performance of the SWE algorithm along with execution dependency. By mapping the instruction associated to their mnemonics into a for loop whose unroll factor was parameterizable, we manage to figure out the latency of main operations involved in the SWE solver (double precision, based on xl compiler). Among dozens of operations, while most of them take only several or tens of cycles (6 cycles for ADD, MUL, and FMA, around 30 cycles for DIV, 40 cycles for *sqrt*()), the TAN and *pow*() operation takes hundreds of cycles. In the SWE solver, the 6 TAN operations from the algorithm can be replaced by 2 TAN, 8 ADD, 4 MUL, and 4 DIV through using the formula $tan(A + B) = (tan(A) + tan(B))/(1 - tan(A) * tan(B))$, which further decreases the computational latency.

Besides the latency reduction approach, SIMD and SMT modification are also adopted on the SWE solver, as discussed in Section 4. However, after these approaches, the performance efficiency only increases to 13.6% (80.35 / 590), which is still significantly lower than the performance efficiency of similar stencil benchmarks. Besides, during the tuning approach, we could find out some tuning methods (such as SIMD) would not come up with satisfying results naturally. Thus, we would like to modify the algorithm to see if a higher performance could be achieved.

6.3 Algorithmic Modification for the SWE Solver

This part aims to further optimize the SWE solver by modifying the solver algorithm. The tuning approach could be summarized as follows: (1) Adjusting the data structure of the SWE solver so as to satisfy the capacity of SIMD registers (Section 6.3.1); (2) Reorganizing the computation sequence of the SWE solver and providing a customized vector grouping strategy, which regroups the data into a hardware-friendly pattern so as to maximize the performance (Section 6.3.2). The experimental results demonstrate both the necessity of algorithm modification and the difference between benchmark tuning and application optimization.

6.3.1 Register-Friendly Data Structure Reconstruction. Algorithm 4 is a simplified code segment that extracted from the SWE algorithm. Among the elements involved in the computation, each member of matrix A is a manually defined structure, which contains three double elements that are employed in the computation of convective fluxes, while each member of matrix B is a double precision data used for calculating source terms. In addition, the size of two matrixes is the same and within each round of computation, five adjacent mesh elements of each matrix are involved.

As the size of a vector-scalar register is 16 bytes, before algorithmic adjustment, SIMD vectorization can only be employed to part of the computation. As demonstrated in the left part of Figure 7, since the VSX instruction set can only load adjacent elements into the same vector, while the computation of s.a and s.b can be paralleled by employing SIMD, the calculation of s.c and q cannot be paralleled since the elements involved when calculating s.c and q are located far away in two different matrixes. That means, among the two matrixes, only 14 elements out of 20 could

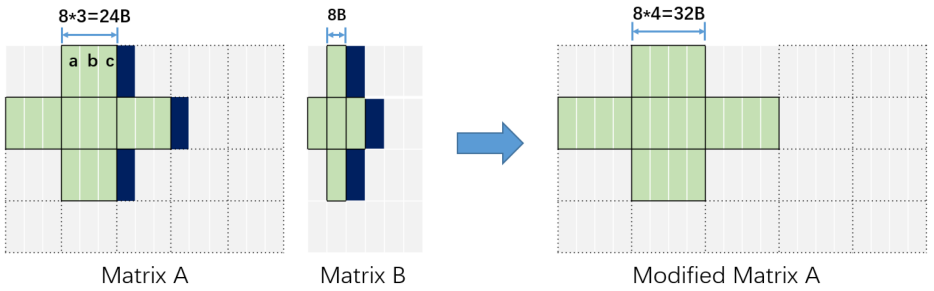


Fig. 7. **Left**: Before adjustment, data of matrix A and matrix B can not properly fit into the 16 bytes vector-scalar registers of POWER8. **Right**: After adjustment, all data could be pre-fetched to vector-scalar registers so as to make the computation more effective.

adopt SIMD vectorization, which means 30% of the computational instructions could not be parallelized. To deal with this issue, we combine the two matrixes together and merge them into a modified matrix A, and each element of A is a data structure that contains four double numbers, as shown in the right part of Figure 7. In this case, all the computation instructions demonstrated in Algorithm 3 could be parallelized via SIMD vectorization.

ALGORITHM 3: Customizing Cache-Friendly Data Structure

Before Adjustment

```
1: s.a← c1*A[j][i-1].a+c2*A[j+1][i-1].a+c3*A[j-1][i-1].a+c4*A[j][i].a-c5*A[j][i-2].a;
2: s.b← c1*A[j][i-1].b+c2*A[j+1][i-1].b+c3*A[j-1][i-1].b+c4*A[j][i].b-c5*A[j][i-2].b;
3: s.c← c1*A[j][i-1].c+c2*A[j+1][i-1].c+c3*A[j-1][i-1].c+c4*A[j][i].c-c5*A[j][i-2].c;
4: q ← c1*B[j][i-1]+c2*B[j+1][i-1]+c3*B[j-1][i-1]+c4*B[j][i]- c5*B[j][i-2];
```

After Adjustment

```
1: s.a←c1*A[j][i-1].a+c2*A[j+1][i-1].a+c3*A[j-1][i-1].a+c4*A[j][i].a-c5*A[j][i-2].a;
2: s.b←c1*A[j][i-1].b+c2*A[j+1][i-1].b+c3*A[j-1][i-1].b+c4*A[j][i].b-c5*A[j][i-2].b;
3: s.c←c1*A[j][i-1].c+c2*A[j+1][i-1].c+c3*A[j-1][i-1].c+c4*A[j][i].c-c5*A[j][i-2].c;
4: s.d←c1*A[j][i-1].d+c2*A[j+1][i-1].d+c3*A[j-1][i-1].d+c4*A[j][i].d-c5*A[j][i-2].d;
```

The register-friendly data structure reconstruction method demonstrated in this part is a representative example of what can be typically done in many parts of the SWE solver or even some other scientific applications ((Fu et al. 2016)); due to the space limitation, we will not come into detail here. By adopting this tuning technique, an 11.2% speedup is achieved in our SWE solver, as shown in Figure 8.

6.3.2 Customized Vector Grouping Strategy. Algorithm 4 is an abstraction of the flux computation part of the SWE solver. While the detailed computation of state reconstruction step shown in Algorithm 3 demonstrates the memory-intensive part of the SWE solver, the simplified Riemann Solver step is the most important computational part that contains more than 900 arithmetic operations in reality.

As demonstrated in Algorithm 4, the computation within the Riemann Solver can be classified into four parts: L (left), R (right), B (bottom), and T (top), where L and R belong to the x-axis, and B and T belong to the y-axis. In this algorithm, variables of the x-axis are calculated at first, and then variables in the y-axis are managed secondly.

To enable the instruction-level parallelization on this part, our first attempt is to vectorize elements within same axis, namely, fusing elements in direction left and right (such as L1 and R1) into a 16-byte vector. In this case, the eight variables adopted in the Algorithm 4 are formulated

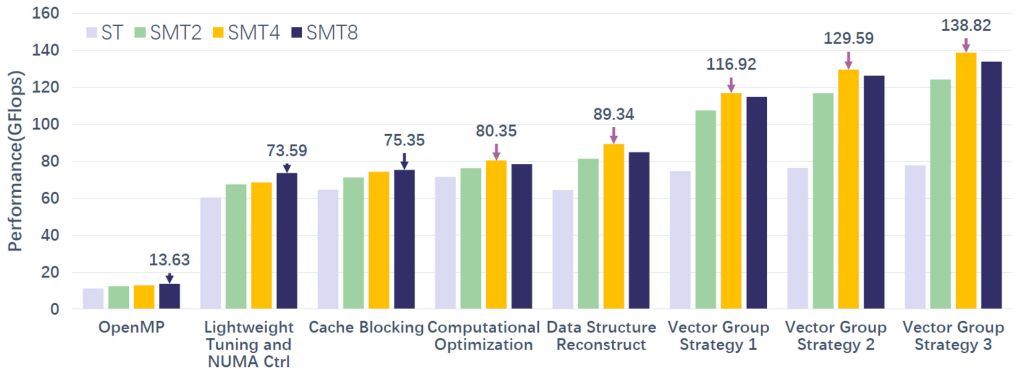


Fig. 8. Performance results of the SWE solver after adopting each tuning technique. (Double precision, grid size is set as 3200*1920, which is far larger than all level of cache capacities.) Latter performance results are inclusive of all the performance gains achieved by adopting previous tuning techniques, except for the three VG strategies (the three VG strategies are independent).

ALGORITHM 4: Simplified Algorithm of Flux Calculation

```
1:  State Reconstruction of x-axis
2:  Riemann Solver of x-axis (simplified)
3:      L←(L1+L2)/L1;
4:      R←(R1+R2)/R1;
5:      gx←(L1+L2)*a+(R1+R2)*a;
6:  State Reconstruction of y-axis
7:  Riemann Solver of y-axis (simplified)
8:      B←(B1+B2)/B2;
9:      T←(T1+T2)/T2;
10:     gy←(B1+B2)*b+(T1+T2)*c;
```

as the following four groups of vector-scalar registers: (L1, R1), (L2, R2), (B1, T1), and (B2, T2). Consequently, the first part (lines 3–4 and 8–9) of the Riemann Solver could be vectorized (demonstrated as *Strategy*1 in Algorithm 5), while the second part (lines 5 and 10) can not apply SIMD vectorization without declaring new VSX registers, as the L1 and R1 are resident in the same VSX register in the previous step. To solve this problem, we adopt more VSX registers as demonstrated in *Strategy*2. In this way, hundreds of computational instructions (84 ADD, 17 SUB, 160 MUL, 24 DIV, 12 SQRT) are parallelized.

To further reduce the new VSX register involved in Strategy 2, we first adjust the computation sequence of the State Reconstruction Step and the Riemann Solver, and then combine the eight variables cross axis as [L1,B2], [L2,B1], [R1,T2], and [R2,T1]. In this case, almost no additional VSX registers are needed in the computation of [gx,gy]; thus, a further performance benefit is able to be achieved. (*Strategy*3)

6.4 Performance Evaluation and Analysis

In this part, we provide the experimental results and related analysis of different tuning techniques for the SWE algorithm based on the IBM S-824L server. Figure 8 demonstrates the performance of the SWE solver after adopting each step of the tuning framework. When the workload is mapped from single core to 24 cores with OpenMP, SWE shows a linear scalability, and this performance serves as a baseline for latter analysis.

ALGORITHM 5: Modified Algorithm of Flux Calculation

Strategy 1: Vectorization along with Axis

```
State Reconstruction of x-axis
Riemann Solver of x-axis (simplified)
    [L,R]←([L1,R1]+[L2,R2])/[L1,R1];
    gx←(L1+L2)*a+(R1+R2)*a;
State Reconstruction of y-axis
Riemann Solver of y-axis (simplified)
    [B,T]←([B1,T1]+[B2,T2])/[B2,T2];
    gy←(B1+B2)*b+(T1+T2)*c;
```

Strategy 2: Further Vectorization of [gx,gy]

```
State Reconstruction of x-axis
Riemann Solver of x-axis (simplified)
    [L,R]←([L1,R1]+[L2,R2])/[L1,R1];
State Reconstruction of y-axis
Riemann Solver of y-axis (simplified)
    [B,T]←([B1,T1]+[B2,T2])/[B2,T2];
    [gx,gy]←([L1,B1]+[L2,B2])*[a,b]+([R1,T1]+[R2,T2])*[a,c];
```

Strategy 3: Vector Grouping Cross Axis

```
State Reconstruction of x-axis
State Reconstruction of y-axis
Riemann Solver of L and B
    [L,B]←([L1,B2]+[L2,B1])/[L1,B2];
Riemann Solver of R and T
    [R,T]←([R1,T2]+[R2,T1])/[R1,T2];
    [gx,gy]←([L1,B2]+[L2,B1])*[a,b]+([R1,T2]+[R2,T1])*[a,c];
```

*square brackets such as "[L,R]" mean variables L and R are put into one 16-byte VSX register, and black square brackets such as "**[L1,R1]**" mean [L1,R1] is a new VSX register that never appeared before.

In Figure 8, the four bars within each tuning method indicate the performance of different SMT modes. Among different tuning techniques, setting SMT as four or eight generally yields better performance than that of ST or SMT=2, which is totally different from the result of stencil computations indicated in Table 4. Such results indicate that within the SWE solver, long-latency operations such as SQRT and TAN combined with execution dependency lead to a large number of empty pipelines and remarkably hit the performance, while employing more than one logical thread within one physical core could release this situation by getting empty pipelines filled.

In addition, the relative performance gap between different SMT models varies when different tuning techniques are adopted. When the SWE solver is far from well-tuned, SMT=8 results in better performance than SMT=4. However, when more tuning techniques are adopted, the performance gap between SMT=4 and SMT=8 gradually disappears and then SMT=4 outperforms SMT=8. The reason is that by adopting systemic tuning techniques, both the operation latency and computation intensity are greatly reduced; as a result, we could use fewer threads to fill the pipelines of the physical core. The fact that the optimal model changes from SMT=8 to SMT=4 proves the effectiveness of our tuning techniques.

Though this work mainly focuses on how to accelerate stencil-based scientific applications on POWER8 system, if we analyze further, at this moment, we could boldly predict that hardware vendors could decrease the maximum SMT mode from 8 to 4, thus, to save the space for more physical cores if the processor is built for scientific computing. As even in the complex SWE solver (part of the program is memory-bound and others are computation-bound, and there are execution dependencies between these parts, which is almost the most complex code organization structure

in the scientific computing field), SMT=4 is enough after adopting tuning techniques presented in this work.

When it comes to first-round tuning methods demonstrated in Figure 8, though the lightweight optimizations are straightforward, remarkable performance benefits are able to be achieved by adopting these optimizations. Adopting cache blocking could increase the L1 cache hit rate and result in an 8.35% speedup. However, the ideal performance is able to be achieved only if we set the size of block as 64KB (size of L1 Dcache), because, though SWE contains large amounts of sophisticated mathematic operations, the mathematic operations nearby the loading operations are all short-latency operations such as ADD, SUB, and MUL. Such feature requires the loading operations to be done within few cycles; otherwise, execution dependency caused by memory loading would hit the performance. Thirdly, based on the latency evaluation of general-used computational operations on POWER8, the latency-reduction algorithm design replaces the long-latency computational operations (namely TAN) with multiple short-latency operations; such an approach decreases the performance hit caused by execution dependency, resulting in another 6.64% speedup.

As for the performance boost provided by customized optimization (namely the algorithmic modification), first of all, the register-friendly data structure reconstruction properly combines the software algorithm with the capacity of vector-scalar registers of POWER8 by reconstructing the input data of the state reconstruction step, leading to a 11.2% speedup. The most remarkable performance boost is achieved by the customized vector grouping strategy. To fully take advantage of the hardware resources, we modify the order of the SWE algorithm and regroup the vector data cross axis. To fully combine the algorithmic features with the hardware characters, three kinds of vector grouping strategies are provided, and according to the experimental result, 30.87%, 45.05%, and 55.38% performance boosts are achieved, respectively. After these steps, the framework is ended since all kinds of tuning techniques (memory optimization, computational optimization and algorithm modification) have been employed. As a result, a 10.18 times speedup is able to be achieved.

7 APPLICATION II: SEISMIC MIGRATION

In this section, we focus on the tuning approach of the reverse time migration (RTM), a more general and simple stencil-based application from seismic modeling.

7.1 Seismic Modeling and RTM Algorithm

Seismic imaging is the basic tool in the oil industry to identify possible reservoir locations. By generating images of the terrain, it is able to explore the geological structures of the earth. The first step for seismic modeling is to generate acoustic waves and record the response at some distance from the source on the surface of the earth. After this approach, we could further rebuild the properties of the propagation media by adopting some single processing algorithm on the recorded data, which is the key step of seismic modeling. While the propagation of waves in the earth is a complex phenomenon that requires a sophisticated wave equation to be modeled accurately, RTM comes out to be one of the popular migration algorithms for seismic modeling (Araya-Polo et al. 2011) (Ortigosa et al. 2008).

As every acoustic shot is introduced in different moments, it is possible to process them independently. Thus, the most external loop of RTM, which sweeps all shots, can be distributed in plenty of nodes or multiple threads. For each shot, we need to prepare the data of the velocity model and the proper set of seismic traces associated with the shot. Due to the same operations inside each shot computation, in this section, we only focus on the RTM kernel, namely the RTM algorithm adopted to process one shot. As demonstrated in Algorithm 6, the output of RTM is the sum of the correlation result of each step which is calculated by the source wave propagation (SWF) and the

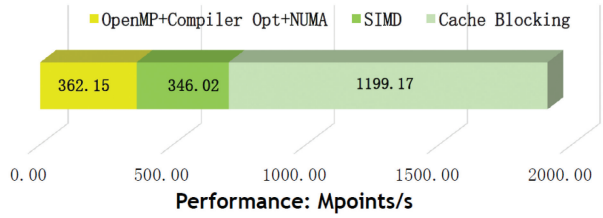
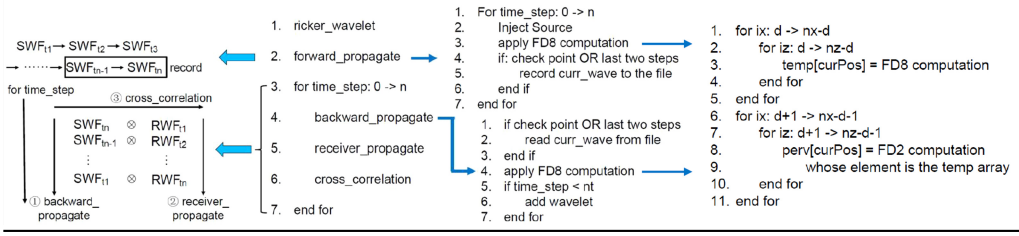


Fig. 9. Demonstration of the RTM performance.

corresponding receiver wave propagation (RWF). However, since the data amount generated by the propagation approach in thousands of timesteps is too huge to be stored in the main memory, we can not record all of the SWF in different timesteps.

ALGORITHM 6: Demonstration of the RTM algorithm with check-point



To deal with this problem, we first record the source propagation results of the last two steps (SWF_{tn} and SWF_{tn-1}, calculation shown in line 2) in main memory. Then, we get the receiver propagation results of the first two steps (RWF_{t1} and RWF_{t2}) and do the correlation of these two steps. After that, we calculate the SWF_{tn-2} in current timestep according to the SWF_{tn} and SWF_{tn-1} by adopting backward propagation (line 4) and get the corresponding RWF_{t3} by performing a receiver propagation (line 5), and then accumulate the correlation results by employing cross correlation (line 6). This approach is repeated until a satisfying result is achieved.

The main challenges of RTM implementation could be classified into two groups: memory access and computation. According to the analysis of Algorithm 6, the computation part of RTM mainly concentrates on the calculation of an FD8-like stencil, as indicated in the right most part of Algorithm 6. During the calculation, since only one direction has the data stored in memory consecutively, access to memory for other directions is very expensive. This feature, together with the long tails of the stencil, makes cache blocking necessary to be adopted to our RTM optimization. Moreover, due to the NUMA on POWER8, a time penalization may be paid if data is not properly distributed among memory banks. In addition, from the computational point of view, we should vectorize the stencil computation in order to make full use of hardware resources.

7.2 Optimization and Performance Analysis

In this part, we demonstrate how to deal with the problems stated above by properly using the optimization techniques presented in the optimization framework.

First, besides applying OpenMP to enable the thread-level parallelism and adopting the optimal combination of compiler options by taking a set of experiments, NUMA control is adopted to decrease the non-local memory access problems. Since the program has good data structure and is able to benefit from SIMD, apparently, two vectorization strategies are employed to balance the hardware usage. According to the experimental results, GPR-oriented vectorization wins again,

and the performance result is demonstrated in Figure 9. After this move, since the application is still bounded by the memory, cache blocking is adopted to decrease the L3 cache miss of POWER processors. Compared with the complex algorithm tuning demonstrated in Section 6, there are few challenges when we adopt the tuning techniques on the RTM program.

The optimal performances after each optimizing step are shown in Figure 9. Compared with the basic OpenMP+NUMA version, a 5.26x speedup is achieved by adopting a series of tuning techniques. We should also note that the testing of different SMT modes is employed during the whole optimizing approach, and according to the experimental results, the best performance shown in the right most bar of Figure 9 is achieved at SMT=2, though the optimal performance of other optimization steps are all obtained at ST (SMT=1) mode.

To evaluate the performance we achieved after adopting our tuning framework, Roofline model (Williams et al. 2009) is involved. Since the RTM algorithm is apparently a memory bound program after adopting SIMD, the achievable peak performance of RTM could be calculated as: $APEAK_{RTM} = FBR_{RTM} * BW_{RTM}$. To simplify the description, we use NUM_{point} and NUM_{ops} to represent the number of points being managed and the number of operations when dealing with each point. According to the hardware architecture of POWER8,

$$BW_{algr} = \begin{cases} BW_{Theoe} * \dfrac{read + write}{read} * \dfrac{2}{3}, & (if\ read > 2 * write) \\ BW_{Theoe} * \dfrac{read + write}{write} * \dfrac{1}{3}, & (otherwise) \end{cases} \tag{1}$$

In addition, 37 read and 5 write need to be done to deal with each point, and the data type is float. From Williams et al. (2009) and the above analysis, we could demonstrate that $FBR_{RTM} = \frac{NUM_{ops}}{(37+5)*4Byte}$ and $BW_{RTM} = 460.8 * \frac{37+5}{37} * \frac{2}{3}$. Thus, the actual performance is $ACPer_{RTM} = NUM_{point} * NUM_{ops}$. As the the NUM_{point} is $\frac{1907.34MPoint/s}{1024} = 1.86$GPoint/s, the ratio of the real performance to the maximum available performance equals to: $\frac{ACPer_{RTM}}{APEAK_{RTM}} = \frac{NUM_{point}*37*6}{460.8} = 89.7\%$. In consideration of the cross-correlation and other operations with the RTM algorithm, this result is quite good and, thus, we could end our tuning framework at this point.

8 CONCLUSION

With the innovation of IBM POWER processors, a number of upcoming supercomputing facilities (such as Summit and Sierra) will use the POWER processors as host processors. However, as current scientific programs are still largely designed for other processors (Intel CPU) and accelerators (GPU, MIC, KNL, etc.), such a transition exposes the lack of tuning guidelines of POWER processors. At this background, this work aims at filling the gap between hardware capability and software performance of the POWER8 processor, as well as facilitating the tuning efforts of the POWER system for stencil-based scientific applications and providing useful guidance.

To achieve this goal, we first demonstrate how to combine the general tuning techniques with the POWER8 hardware architecture through the optimization approach of three representative stencil benchmarks. By properly using the five optimization methods, our performance based on POWER8 is two times better than the performance of contemporary mainstream processors (such as Intel E5-2697 v2). These results demonstrate the effect of our tuning guidelines and prove that POWER8 is an ideal platform for running stencil-based scientific applications.

Based on these optimization guidelines and their evaluation, we further provide a performance tuning framework aimed at guiding the tuning approach of stencil-based applications on POWER processors so as to facilitate the performance tuning approach for programmers. After such approach, we further choose two typical scientific applications (namely, the similar kernels of the

winner program of the Gordon Bell Prize 2016 and 2017) to show how to deal with the challenges when we optimize real-world applications on POWER processors. Benefits from the proper algorithmic adjustment (even five versions on the atmospheric modeling program) and remarkable performance boost (10.18x, 5.26x speedup) is achieved. Such a process demonstrates the way of combining the hardware-oriented tuning strategies with the application algorithms, and also proves that our tuning framework works well for both simple and complex stencil-based applications.

To summarize, this work demonstrates that though the raw performance of POWER8 may not be as good as other mainstream processors (Mericas et al. 2015; Datta et al. 2008), with proper manual design, the POWER8 system could be one of the most powerful multi-core CPU processors for scientific computing, as described in Section 5.2. As we fill the gap between the hardware capability and software performance of POWER8, scientific application programmers are able to boost their programs by employing framework or guidelines presented in this article. We could predict that POWER CPU is also a promising candidate that can serve as the host processor to construct the hybrid architecture, such as the POWER-GPU, or POWER-FPGA schemes. These ideas will be studied in future.

ACKNOWLEDGMENTS

The authors are grateful to the reviewers for their valuable comments that have greatly improved the article. The authors would also like to thank Dr. Song Yu and Peng Hongbo of IBM China Systems and Technology Laboratory for providing the testbed as well as the technical support.

REFERENCES

Andrew V. Adinetz, Paul F. Baumeister, Hans Böttiger, Thorsten Hater, Thilo Maurer, Dirk Pleiter, Wolfram Schenck, and Sebastiano Fabio Schifano. 2014. Performance evaluation of scientific applications on POWER8. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 24–45.

Mauricio Araya-Polo, Javier Cabezas, Mauricio Hanzich, Miquel Pericas, Felix Rubio, Isaac Gelado, Muhammad Shafiq, Enric Morancho, Nacho Navarro, Eduard Ayguade, and others. 2011. Assessing accelerator-based HPC reverse time migration. *IEEE Transactions on Parallel and Distributed Systems* 22, 1 (2011), 147–162.

Mike Ashworth, Jianping Meng, Vedran Novakovic, and Sersi Siso. 2016. Early application performance at the Hartree Centre with the OpenPOWER architecture. In *International Conference on High Performance Computing*. Springer, 173–187.

Alexander Berreth, Benedetto Risio, Markus Bühler, Benedikt Anlauf, and Pascal Vezolle. 2016. Performance of the 3D combustion simulation code RECOM®-AIOLOS on IBM® POWER8® architecture. In *Proceedings of the International Conference on High Performance Computing*. Springer, 286–292.

Shirley Browne, Jack Dongarra, Nathan Garner, George Ho, and Philip Mucci. 2000. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications* 14, 3 (2000), 189–204.

Juan M. Cebrián, José M. Cecilia, Mario Hernández, and José M. García. 2017. Code modernization strategies to 3-D Stencil-based applications on Intel Xeon Phi: KNC and KNL. *Computers & Mathematics with Applications* 74, 10 (2017), 2557–2571.

Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE Press, 4.

Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. 2003. The LINPACK benchmark: Past, present and future. *Concurrency and Computation: Practice and experience* 15, 9 (2003), 803–820.

Timothée Ewart, Stuart Yates, Francesco Cremonesi, Pramod Kumbhar, Felix Schürmann, and Fabien Delalondre. 2015a. Performance evaluation of the IBM POWER8 architecture to support computational neuroscientific application using morphologically detailed neurons. In *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*. ACM, 1.

Timothée Ewart, Stuart Yates, Francesco Cremonesi, Pramod Kumbhar, Felix Schürmann, and Fabien Delalondre. 2015b. Performance evaluation of the IBM POWER8 architecture to support computational neuroscientific application using

morphologically detailed neurons. In *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*. ACM, 1.

Eric J. Fluhr, Joshua Friedrich, Daniel Dreps, Victor Zyuban, Gregory Still, Christopher Gonzalez, Allen Hall, David Hogenmiller, Frank Malgioglio, Ryan Nett, and others. 2014. 5.1 POWER8 TM: A 12-core server-class processor in 22nm SOI with 7.6 Tb/s off-chip bandwidth. In *Proceedings of the 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 96–97.

Joshua Friedrich, Hung Le, William Starke, Jeff Stuechli, Balaram Sinharoy, Eric J. Fluhr, Daniel Dreps, Victor Zyuban, Gregory Still, Christopher Gonzalez, and others. 2014. The POWER8 TM processor: Designed for big data, analytics, and cloud environments. In *Proceedings of the 2014 IEEE International Conference on IC Design & Technology (ICICDT)*. IEEE, 1–4.

Haohuan Fu, Lin Gan, Robert G. Clapp, Huabin Ruan, Oliver Pell, Oskar Mencer, Michael Flynn, Xiaomeng Huang, and Guangwen Yang. 2014. Scaling reverse time migration performance through reconfigurable dataflow engines. *IEEE Micro* 34, 1 (2014), 30–40.

Haohuan Fu, Conghui He, Bingwei Chen, Zekun Yin, Zhenguo Zhang, Wenqiang Zhang, Tingjian Zhang, Wei Xue, Weiguo Liu, Wanwang Yin, and others. 2017a. 18.9-Pflops nonlinear earthquake simulation on Sunway TaihuLight: Enabling depiction of 18-Hz and 8-meter scenarios. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2.

Haohuan Fu, Junfeng Liao, Nan Ding, Xiaohui Duan, Lin Gan, Yishuang Liang, Xinliang Wang, Jinzhe Yang, Yan Zheng, Weiguo Liu, and others. 2017b. 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, 1.

Haohuan Fu, Jingheng Xu, Lin Gan, Chao Yang, Wei Xue, Wenlai Zhao, Wen Shi, Xinliang Wang, and Guangwen Yang. 2016. Unleashing the performance potential of CPU-GPU platforms for the 3D atmospheric Euler solver. In *Proceedings of the 2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 41–49.

L. Gan, H. Fu, W. Luk, and C. Yang. 2013. Accelerating solvers for global atmospheric equations through mixed-precision data flow engine. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. 1–6.

Lin Gan, Haohuan Fu, Wayne Luk, Chao Yang, Wei Xue, and Guangwen Yang. 2017. Solving mesoscale atmospheric dynamics using a reconfigurable dataflow architecture. *IEEE Micro* 37, 4 (2017), 40–50.

L. Gan, H. Fu, O. Mencer, W. Luk, and G. Yang. 2016. Chapter four: Data flow computing in geoscience applications. *Advances in Computers* (2016).

Lin Gan, Haohuan Fu, Wei Xue, Yangtong Xu, Chao Yang, Xinliang Wang, Zihong Lv, Yang You, Guangwen Yang, and Kaijian Ou. 2014a. Scaling and analyzing the stencil performance on multi-core and many-core architectures. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 103–110.

Lin Gan, Haohuan Fu, Chao Yang, Wayne Luk, Wei Xue, Oskar Mencer, Xiaomeng Huang, and Guangwen Yang. 2014b. A highly-efficient and green data flow engine for solving Euler atmospheric equations. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. 1–6.

Brian Hall, Ryan Arnold, Peter Bergner, Wainer dos Santos Moschetta, Robert Enenkel, Pat Haugen, Michael R. Meissner, Alex Mericas, Philipp Oehler, Berni Schiefer, and others. 2014. *Performance Optimization and Tuning Techniques for IBM Processors, Including IBM POWER8*. IBM Redbooks.

IBM, Wikipedia, and Anandtech. References of IBM POWER8. Retrieved from https://en.wikipedia.org/wiki/POWER8, https://en.wikipedia.org/wiki/Power_Architecture, https://www.anandtech.com/show/10435/assessing-ibms-power8-part-1/2.

Middle East IBM Europe and Africa Software Announcement. 2014. IBM XL C/C++ for Linux, V13.1 delivers IBM POWER8 exploitation and enhancements to improve performance and language standards conformance. Retrieved April 2014 from https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?infotype=an&subtype=ca&appname=gpateam&supplier=877&letternum=ENUSZP14-0196.

Tsuyoshi Ichimura, Kohei Fujita, Pher Errol Balde Quinay, Lalith Maddegedara, Muneo Hori, Seizo Tanaka, Yoshihisa Shizawa, Hiroshi Kobayashi, and Kazuo Minami. 2015. Implicit nonlinear wave simulation with 1.08 T DOF and 0.270 T unstructured finite elements to enhance comprehensive earthquake simulation. In *Proceedings of the 2015 SC-International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

Intel, Wikipedia, and stuffedcow. References of Intel E5-2697(v2). Retrieved from https://ark.intel.com/products/75283, http://blog.stuffedcow.net/2013/05/measuring-rob-capacity/, https://en.wikipedia.org/wiki/Ivy_Bridge_%28microarchitecture%29, https://en.wikipedia.org/wiki/List_of_Intel_CPU_microarchitectures, https://en.wikipedia.org/wiki/List_of_Intel_Xeon_microprocessors"Ivy_Bridge-EP"_(22_nm)_Efficient_Performance_2.

Jim Jeffers and James Reinders. 2015. *High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches*. Morgan Kaufmann.

Peter Johnsen, Mark Straka, Melvyn Shapiro, Alan Norton, and Thomas Galarneau. 2013. Petascale WRF simulation of Hurricane Sandy: Deployment of NCSA's cray XE6 blue waters. In *Proceedings of the 2013 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 1–7.

Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. 2010. An auto-tuning framework for parallel multicore stencil computations. In *Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 1–12.

Lawrence Livermore National Laboratory. 2017a. Livermore's next advanced technology high performance computing system. Retrieved from https://computation.llnl.gov/computers/sierra.

Oak Ridge National Laboratory. 2017b. Oak Ridge National Laboratory's next high performance supercomputer. Retrieved from https://www.olcf.ornl.gov/summit/.

Xing Liu, Daniele Buono, Fabio Checconi, Jee W. Choi, Xinyu Que, Fabrizio Petrini, John A. Gunnels, and Jeff A. Stuecheli. 2016. An early performance study of large-scale POWER8 SMP systems. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 263–272.

John D. McCalpin. 1995. STREAM: Sustainable memory bandwidth in high performance computers. Retrieved from https://www.cs.virginia.edu/stream/.

A. Mericas, N. Peleg, Lorena Pesantez, S. B. Purushotham, P. Oehler, C. A. Anderson, B. A. King-Smith, M. Anand, J. A. Arnold, B. Rogers, and others. 2015. IBM POWER8 performance features and evaluation. *IBM Journal of Research and Development* 59, 1 (2015), 6–1.

Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 2010. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In *Proceedings of the 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 1–13.

Francisco Ortigosa, Qingbo Liao, Antoine Guitton, and Wenying Cai. 2008. Speeding up RTM velocity model building beyond algorithmics. In *SEG Technical Program Expanded Abstracts 2008*. Society of Exploration Geophysicists, 3219–3223.

István Z. Reguly, Abdoul-Kader Keita, and Michael B. Giles. 2015. Benchmarking the IBM Power8 processor. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*. IBM Corp., 61–69.

István Z. Reguly, Abdoul-Kader Keita, Rafik Zurob, and Michael B. Giles. 2016. High performance computing on the IBM Power8 platform. In *International Conference on High Performance Computing*. Springer, 235–254.

Gabriel Rivera and Chau-Wen Tseng. 2000. Tiling optimizations for 3D scientific computations. In *Proceedings of the ACM/IEEE 2000 Conference on Supercomputing*. IEEE, 32–32.

Johann Rudi, A. Cristiano I. Malossi, Tobin Isaac, Georg Stadler, Michael Gurnis, Peter W.J. Staar, Yves Ineichen, Costas Bekas, Alessandro Curioni, and Omar Ghattas. 2015. An extreme-scale implicit solver for complex PDEs: Highly heterogeneous flow in earth's mantle. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 5.

Balaram Sinharoy, J.A. Van Norstrand, Richard J. Eickemeyer, Hung Q. Le, Jens Leenstra, Dung Q. Nguyen, B. Konigsburg, K. Ward, M.D. Brown, José E. Moreira, and others. 2015. IBM POWER8 processor core microarchitecture. *IBM Journal of Research and Development* 59, 1 (2015), 2–1.

R. Smith, P. Jones, B. Briegleb, F. Bryan, G. Danabasoglu, J. Dennis, J. Dukowicz, C. Eden, B. Fox-Kemper, P. Gent, and others. 2010. *The Parallel Ocean Program (POP) Reference Manual. Los Alamos National Lab Technical Report* 141.

John E. Stone, Antti-Pekka Hynninen, James C. Phillips, and Klaus Schulten. 2016. Early experiences porting the NAMD and VMD molecular simulation and analysis software to GPU-accelerated OpenPOWER platforms. In *Proceedings of the International Conference on High Performance Computing*. Springer, 188–206.

Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and West Pomeranian. 2011. Impact of system and cache bandwidth on stencil computations across multiple processor generations. In *Proceedings of the Workshop on Applications for Multi- and Many-Core Processors (A4MMC) at ISCA, 2011*.

Carl A. Waldspurger. 1996. *Lottery and Stride Scheduling: Flexibile Proportional-share Resource Management*. Massachusetts Institute of Technology. 1 page.

Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM* 52, 4 (2009), 65–76.

Jingheng Xu, Haohuan Fu, Lin Gan, Yu Song, Hongbo Peng, Wen Shi, and Guangwen Yang. 2016a. Evaluating the POWER8 architecture through optimizing stencil-based algorithms. In *Trustcom/BigDataSE/ISPA, 2016 IEEE*. IEEE, 1374–1381.

Jingheng Xu, Haohuan Fu, Lin Gan, Chao Yang, Wei Xue, Shizhen Xu, Wenlai Zhao, Xinliang Wang, Bingwei Chen, and Guangwen Yang. 2016b. Generalized GPU acceleration for applications employing finite-volume methods. In *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 126–135.

Wei Xue, Chao Yang, Haohuan Fu, Xinliang Wang, Yangtong Xu, Lin Gan, Yutong Lu, and Xiaoqian Zhu. 2014. Enabling and scaling a global shallow-water atmospheric model on tianhe-2. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 745–754.

Chao Yang, Wei Xue, Haohuan Fu, Lin Gan, Linfeng Li, Yangtong Xu, Yutong Lu, Jiachang Sun, Guangwen Yang, and Weimin Zheng. 2013. A peta-scalable CPU-GPU algorithm for global atmospheric simulations. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 1–12.

Chao Yang, Wei Xue, Haohuan Fu, Hongtao You, Xinliang Wang, Yulong Ao, Fangfang Liu, Lin Gan, Ping Xu, Lanning Wang, and others. 2016. 10M-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC16*. IEEE, 57–68.

Received February 2018; revised July 2018; accepted July 2018