Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Programming bare-metal accelerators with heterogeneous threading models: a case study of Matrix-3000[*]

Jianbin FANG[†§1], Peng ZHANG[†§1], Chun HUANG[†‡1], Tao TANG[1],
Kai LU[1], Ruibo WANG[1], Zheng WANG[2]

[1]*College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China*
[2]*School of Computing, University of Leeds, Leeds LS2 9JT, UK*
[†]E-mail: j.fang@nudt.edu.cn; zhangpeng13a@nudt.edu.cn; chunhuang@nudt.edu.cn

Received Aug. 27, 2022; Revision accepted Oct. 19, 2022; Crosschecked Mar. 8, 2023

Abstract: As the hardware industry moves toward using specialized heterogeneous many-core processors to avoid the effects of the power wall, software developers are finding it hard to deal with the complexity of these systems. In this paper, we share our experience of developing a programming model and its supporting compiler and libraries for Matrix-3000, which is designed for next-generation exascale supercomputers but has a complex memory hierarchy and processor organization. To assist its software development, we have developed a software stack from scratch that includes a low-level programming interface and a high-level OpenCL compiler. Our low-level programming model offers native programming support for using the bare-metal accelerators of Matrix-3000, while the high-level model allows programmers to use the OpenCL programming standard. We detail our design choices and highlight the lessons learned from developing system software to enable the programming of bare-metal accelerators. Our programming models have been deployed in the production environment of an exascale prototype system.

Key words: Heterogeneous computing; Parallel programming models; Programmability; Compilers; Runtime systems
https://doi.org/10.1631/FITEE.2200359 **CLC number:** TP315

1 Introduction

Heterogeneous many-core processors are now commonplace in computer systems (Owens et al., 2005, 2008). The combination of a host central processing unit (CPU) with a specialized accelerator (e.g., a general-purpose graphics processing unit (GPGPU), field programmable gate array (FPGA), digital signal processor (DSP), or neural processing unit (NPU)) is shown to deliver orders of magnitude performance improvement over traditional homogeneous CPU setups (Patterson, 2018). The increasing importance of heterogeneous many-core architectures can be seen in the TOP500 (https://www.top500.org/lists/top500/) and Green500 (https://www.top500.org/lists/green500/) lists, where a large number of supercomputers are integrated with CPUs and accelerators (Liao et al., 2018). Indeed, the heterogeneous many-core architecture is widely seen as the building block for next-generation supercomputers.

The potential of accelerators can be unlocked only if the software can make good use of the hardware (Fang et al., 2011; Shen et al., 2012). Writing and optimizing code for many-core accelerators is challenging for many application developers. This is because the current hardware architecture and

[§] These two authors contributed equally to this work
[‡] Corresponding author
[*] Project supported by the National Key Research and Development Program of China (No. 2021YFB0300101), the National Natural Science Foundation of China (No. 61972408), and the UK Royal Society International Collaboration Grant
ORCID: Jianbin FANG, https://orcid.org/0000-0003-3542-4869; Peng ZHANG, https://orcid.org/0000-0001-8364-9793; Chun HUANG, https://orcid.org/0000-0002-0317-8192

programming model of accelerators significantly differ from those of the conventional multi-core processors (Perez et al., 2007). This change has, by default, shifted the burden of developing a suitable software framework onto programmers and compilers (Kudlur and Mahlke, 2008). In particular, programmers have to manage the hardware heterogeneity, parallelism, and a complex, distributed memory hierarchy (Zhai and Chen, 2018).

In this paper, we share our experience of designing and implementing a programming model and its supporting compiler and runtime system for the Matrix-3000 (also coined as MT-3000) heterogeneous many-core accelerator (Section 2). This accelerator is designed to be a building block for next-generation exascale prototype supercomputers (Lu et al., 2022). While providing potential high performance, MT-3000 has a complex memory hierarchy and processor organization. Furthermore, as no operating system (OS) runs on the accelerators, it is highly challenging to debug and manage parallel threads running on them. In a nutshell, the architecture design of MT-3000 poses a range of challenges to the low-level system software design.

To support software development for MT-3000, we have developed a piece of full-stack system software (Section 3) from scratch, where we focus on two programming modules: the `hthreads` low-level programming interface (Section 4) and the `MOCL3` OpenCL compiler (Section 5). Specifically, we develop a low-overhead high-availability heterogeneous programming interface (`hthreads`). At the core of `hthreads` is a heterogeneous threading model introduced for the bare-metal MT-3000 accelerator. The `hthreads` interface consists of the general-purpose (GP) zone side application programming interfaces (APIs) and the acceleration (ACC) zone side APIs. On one hand, `hthreads` exposes as many performance-related architecture features as we can, aiming to fully tap its computing potentials. On the other hand, we introduce the threading model to hide the metal-related uses such as the native direct memory access (DMA) usage. At a higher level, we provide the implementation of the OpenCL standard parallel programming interface (`MOCL3`) for the MT-3000 architecture. It follows the programming specification of OpenCL version 1.2. While ensuring to explore the computing potential of MT-3000, `MOCL3` can be effectively compatible with OpenCL legacy code and significantly improve programmability. With these two programming interfaces, we aim to achieve a balance among performance, programmability, and portability for our bare-metal accelerator.

Both `hthreads` and `MOCL3` have been deployed in the production environment of an exascale prototype system with the MT-3000 executing `hthreads` and `MOCL3` optimized code at any time. We showcase the performance of the developed system on micro-benchmarks and matrix multiplications (Section 6). We hope that the experience presented in this paper can provide new insights into the development of future high-performance accelerators and their programming systems.

2 Matrix-3000 architecture

This section describes the hardware architecture design of the MT-3000 accelerator.

2.1 Heterogeneous multi-zones

As depicted in Fig. 1, MT-3000 implements a multi-zone microarchitecture with 16 CPU cores, 96 control cores, and 1536 acceleration cores. The CPU cores form a GP zone, while the control cores and acceleration cores form an ACC zone. The ACC zone is then equally divided into four autonomous acceleration clusters. Each cluster has 24 control cores, 384 acceleration cores, and on-chip global shared memory (GSM), high-bandwidth shared memory (HBSM), and off-chip double data rate (DDR) memory. The GP zone CPU cores run at 2.0 GHz, while the ACC zone cores operate at 1.2 GHz and can deliver a total of 11.6 Tflops double-precision

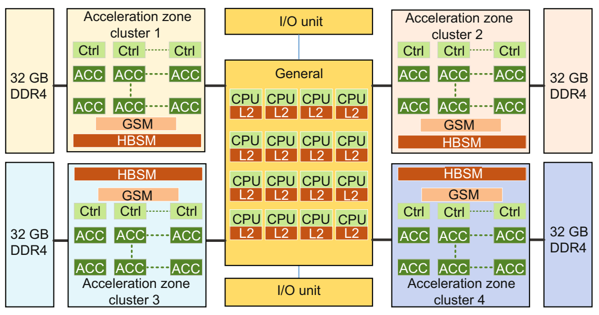

Acceleration zone cluster 1
Acceleration zone cluster 2
Acceleration zone cluster 3
Acceleration zone cluster 4
32 GB DDR4
I/O unit
General
CPU
L2
Ctrl
ACC
GSM
HBSM

Fig. 1 Overview of the Matrix-3000 architecture (DDR: double data rate; Ctrl: control; ACC: acceleration; GSM: global shared memory; HBSM: high-bandwidth shared memory; I/O: input/output; CPU: central processing unit)

performance with a power efficiency of 45.4 Gflops per watt. The four ACC zone clusters can run independent of each other.

An OS runs in the GP zone. Different from the GP zone, the acceleration cluster is a bare-metal device, with no support of OS, and all hardware resources need to be managed by user programs. The CPU cores in the GP zone are capable of managing the overall task execution, running the OS, and processing general-purpose tasks, while the ACC zone is designed for computation-intensive tasks. The CPU and the accelerator have different accessing scopes of the memory hierarchy, which are detailed in the following subsection.

2.2 Hybrid memory hierarchy

MT-3000 is featured with a hybrid memory hierarchy. Processing cores in the GP and ACC zones have different memory accessing scopes.

1. General-purpose zone

Each of the 16 CPUs in the GP zone has its own L1 and L2 cache. The CPU cores are connected through a mesh-based on-chip network (NoC) that achieves cache coherency. The size of the L2 cache is 512 KB, and it is organized in a 16-way set associated with a 64-byte cacheline. The L1 cache adopts an inclusive policy. Therefore, when an L2 data element is requested by other CPUs, the L2 cache controller will have to retrieve the newest values from its corresponding L1 cache if the data are dirty. The CPUs support optional prefetch; i.e., upon a cache miss, 0, 2, 4, or 8 cachelines would be prefetched into L2.

2. Acceleration zone

As shown in Fig. 1, each cluster in the ACC zone has its own GSM, HBSM, and DDR memories. GSM and HBSM are private to the cluster, but are shared by all the control and acceleration cores within the same acceleration cluster. To reduce memory conflicts, both HBSM and GSM are organized into multiple banks. Combined with direct memory access, such a multi-banked organization provides flexible support for runtime data management.

The CPU core can access the entire HBSM/GSM and the DDR space located in different acceleration clusters. By contrast, acceleration cores can access GSM, HBSM, and DDR only within their acceleration cluster. Data transfers across different acceleration clusters have to be managed by CPUs.

2.3 Acceleration array organization

2.3.1 Combined very long instruction word (VLIW) and acceleration array

Fig. 2 shows that each MT-3000 acceleration cluster has 24 acceleration arrays, each of which further has one control core and 16 acceleration cores. In the acceleration array, the 16 acceleration cores are driven by one single instruction stream and work in a lock-step manner. The single instruction stream is handled by the control core. The acceleration array is the main source for data-level parallelism (DLP), which is commonly seen in high-performance computing (HPC) workloads.

Furthermore, each acceleration core uses a VLIW organization, having three multiply-and-accumulate (MAC) units, one integer execution unit (IEU), and two load/store units. At most six instructions can be packed and issued simultaneously to an acceleration core within a cycle. Each MAC unit supports both fixed and floating-point multiplication-and-accumulation. The MAC unit supports half-, single-, and double-precision floating-point operations. IEU can support both bitwise and integer operations. By combining VLIW and acceleration array organization, we can exploit both data- and instruction-level parallelism.

2.3.2 On-chip memory design

MT-3000 uses a high-bandwidth on-chip memory design, i.e., scalar memory (SM) and array memory (AM), which is shown in Fig. 2. SM is private to

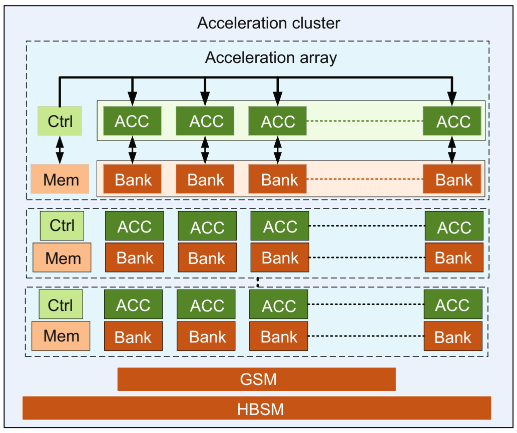


Fig. 2 Organization of an acceleration cluster (Ctrl: control; Mem: scalar memory; ACC: acceleration; GSM: global shared memory; HBSM: high-bandwidth shared memory)

a control core and AM is shared by the 16 acceleration cores. AM supports at most two loads/stores on each acceleration core. The data types of AM load/store include half-word (32-bit), word (64-bit), and double-word (128-bit). Thus, AM can provide at most 512 bytes ($16 \times 2 \times 128$ bits) to 16 acceleration cores. Each SM buffer is of 64 KB, which is private to a control core, and each AM buffer is of 768 KB, which is private to the 16 acceleration cores. Note that the AM buffer and the SM buffer are located at the same level.

To summarize, MT-3000 is a bare-metal heterogeneous processor with a complex core and memory organization. Such hardware design provides the potential for high performance by giving a large degree of flexibility to the system software to optimize data placement, thread communications, and parallel computation. However, this architectural design requires having an effective programming model to lower the programming difficulties. Our work will focus on addressing the programming issues of MT-3000 and similar accelerators.

3 MT-3000 programming stack

Fig. 3 gives an overview of our four-layer software stack designed for MT-3000. A standard Linux OS runs on the CPU. The OS manages the interactions between CPUs and the bare-metal accelerator clusters in the ACC zone through a device driver. The MT-3000 compiler, `m3cc`, translates C code into executable binaries to run on MT-3000. As part of the compilation toolchain, `libMT` provides a low-level interface for the runtime to manage accelerators, and `HPML` is a high-performance math library specifically optimized for MT-3000.

At the core of this work are the two programming interfaces that we have developed for MT-3000.

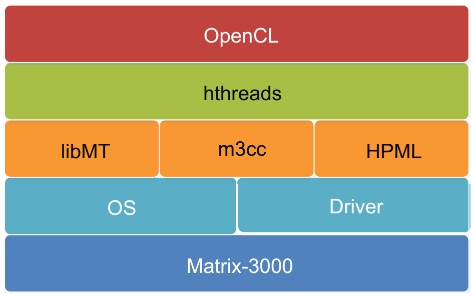


Fig. 3 The MT-3000 programming stack (OS: operating system)

The `hthreads` programming model provides a heterogeneous threading model for MT-3000. Built upon `hthreads`, `MOCL3` implements the OpenCL heterogeneous programming standard of version 1.2 for MT-3000. In a nutshell, `hthreads` and `MOCL3` are built on top of the lower-level components of the software stack of MT-3000.

3.1 The m3cc compiler

As a key component of the MT-3000 software stack, we develop a low-level, in-house compiler (`m3cc`), which translates C99 compatible programs into executable MT-3000 binaries. `m3cc` is a cross-compiler that runs on the general-purpose CPU of MT-3000 to generate binaries for the accelerator. In addition to standard C, `m3cc` supports vectorization intrinsics and embedded assembly codes. `m3cc` is integrated with the assembler (`m3cc-as`) and the linker (`m3cc-ld`), which form a complete compiling toolchain for MT-3000.

3.2 Low-level software interface

The `libMT` library acts as a low-level software interface for managing the interactions between CPUs and accelerators. This library first provides the functions of managing the shared buffers and data transfers between CPUs and accelerators. Given that accelerators can use only physical addresses and CPUs use virtual addresses, `libMT` has to perform address translation between them based on the lower-level driver module. Then, `libMT` loads a program image and its kernel argument data onto a predefined location, and fires the accelerators for kernel execution. As there are caches on the CPU side, `libMT` also provides interfaces to maintain data consistency between CPUs and accelerators by invalidating the data cachelines when needed.

3.3 High-performance math library

`HPML` is a bundle of mathematical libraries, including libm, basic linear algebra subprograms (BLASs), sparse BLASs (SparseBLASs), fast Fourier transform (FFT), and many others. The math libraries are highly optimized by experts for the accelerators of MT-3000, aiming to fully tap its computing potentials of MT-3000. They are typically hardcoded kernels in assemblies. On the host side, they are implemented in `libMT` or `hthreads` to manage the interactions between the GP zone and the ACC zone.

4 The hthreads programming interface

4.1 Design overview

To avoid users having to directly deal with the underlying hardware, and to improve programmability, we present a low-overhead high-availability heterogeneous threads (`hthreads`) programming interface.

Fig. 4 shows that `hthreads` consists of GP zone side APIs (host APIs) and ACC zone side APIs (device APIs). In general, `hthreads` takes the GP side as *host*, and takes each acceleration cluster as a *compute device*. On the host side, we provide APIs to manage devices, program images, threads, and shared resources. At the core of `hthreads` is the introduction of the *threading* concept. Thus, programmers can use the logical threading instance, rather than the physical-cores-related concepts. On the device side, we provide APIs to manage thread parallelism, synchronization, data movements between on-chip and off-chip buffers, and the orchestration of the 16 acceleration cores with vector data types and intrinsics.

Below we show example codes in `hthreads`. Fig. 5 shows a code example for vector addition ($\boldsymbol{C}[\,] = \boldsymbol{A}[\,] + \boldsymbol{B}[\,]$) in `hthreads`. We see that `hthreads` uses `hthread_dev_open` and `hthread_dev_close` to switch the device on and off respectively, and conducts necessary initialization. We use `hthread_dat_load` to load the compiled kernel image into the predefined location (line 3). Then it allocates buffers for a specific device with `hthread_malloc` (lines 4–6), and

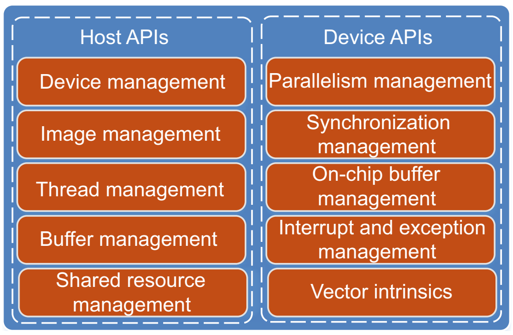


Fig. 4 Overview of the hthreads interfaces (API: application programming interface)

they are freed before the program exits (line 10). Programmers have to explicitly specify the buffer size and properties (`HT_MEM_RO`, `HT_MEM_WO`, or `HT_MEM_RW`). After preparing the arguments, we create a *thread group* and launch kernel execution with `hthread_group_create` (line 7). When coding kernels, we use a qualifier `__global__`. To manage threads on the device side, we use `get_thread_id` to identify a thread, which is pinned to an acceleration array (Fig. 2). We then use `vector_{malloc|free}` and `vector_{load|store}` APIs for AM management and data movements. These interfaces encapsulate the hardware details of the DMA units with abstractions.

```
 1  /* main entry for vector addition */
    hthread_dev_open(dev_id);
 3  hthread_dat_load(dev_id, "vadd_kernel.dat");
    long *A = hthread_malloc(dev_id, size,
        ↪ HT_MEM_RO);
 5  long *B = hthread_malloc(dev_id, size,
        ↪ HT_MEM_RO);
    long *C = hthread_malloc(dev_id, size,
        ↪ HT_MEM_RW);
 7  int tg_id = hthread_group_create(dev_id,
        ↪ num_threads, "add_vector", 2, 3,
        ↪ args);
    hthread_group_wait(tg_id);
 9  hthread_group_destroy(tg_id);
    hthread_free(A); /* the same for B and C */
11  hthread_dev_close(dev_id);
    /* borderline between host and device codes
        ↪ */
13  __global__ void add_vector(unsigned long
        ↪ length, long *A, long *B, long *C){
        int tid = get_thread_id();
15      long i = 0, b = 0;
        long step_size = 16 * 1024;
17      long num_steps = length/step_size;
        unsigned long offset = tid * length;
19      lvector long * src1 = vector_malloc(
            ↪ step_size * sizeof(long));
        lvector long * src2 = vector_malloc(
            ↪ step_size * sizeof(long));
21      lvector long * dst = vector_malloc(
            ↪ step_size * sizeof(long));
        for(b = 0; b < num_steps; b++){
23          vector_load(&A[offset], src1,
                ↪ step_size * sizeof(long));
            vector_load(&B[offset], src2,
                ↪ step_size * sizeof(long));
25          for (i = 0; i < step_size / 16; i
                ↪ ++)
                dst[i] = src1[i] + src2[i];
27          vector_store(dst, &C[offset],
                ↪ step_size * sizeof(long));
            offset = offset + step_size;
29      }
        vector_free(src1);
31      vector_free(src2);
        vector_free(dst);
33  }
```

Fig. 5 An example in hthreads for vector addition

4.2 Implementation

Fig. 6 shows the conceptual framework of `hthreads`. The modules above the dashed line are the interface provided to users, and the internal implementation function modules are below the dashed line. GP and ACC zones communicate through interrupt and shared memory.

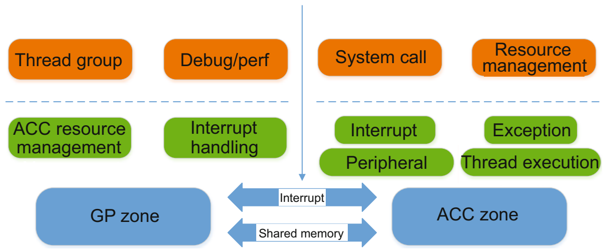


Fig. 6 The hthreads implementation modules (perf: performance analysis; ACC: acceleration; GP: general-purpose)

4.2.1 Memory layout

As acceleration clusters have no support of OS, `hthreads` has to prepare execution contexts for kernels. Kernel execution can be treated as an independent process, which has its own process space. Before kernel execution, `hthreads` will load code segments and data segments into predefined memory locations, prepare the stack register, set the entry point, and pass parameters to kernel functions according to the calling conversion (line 3 of Fig. 5). Thus, we need to set up the memory layout for them. For the use convenience, we also have to map HBSM, GSM, AM, and SM into the process space.

Due to the complex memory hierarchy, the mapping space of the memory layout is particularly large. We can choose to place the scalar stack on DDR, GSM, HBSM, or SM. The code and data segments can also be mapped onto DDR, GSM, or HBSM. When we put the scalar stack on SM, the instruction latency will be short. However, SM has limited capacity and it could be too small to hold large codes. Alternatively, when we put the scalar stack on DDR, the latency will be large. Therefore, we have to make a tradeoff between performance and buffer capacity.

For the acceleration cores of the ACC zone, we place the heap and stack on the private vector memory of the acceleration array. Note that the stack space grows downwards and the heap space grows upwards. As we do not support thread switching, a thread will occupy an entire acceleration array in an exclusive manner until it exits.

4.2.2 Communication between GP and ACC zones

Since `libMT` is the library of the GP zone, there is no way to communicate between GP and ACC zones. We need to implement the communication driver in `hthreads`. We choose not to use a daemon process to act as a management process in the ACC zone. Instead, the process/thread running on each control core has to manage its hardware resource independently. This means that the GP zone needs to communicate with all the used control cores.

We build the communication driver based on the interrupt system to support real-time interactions between GP and ACC zones. On the ACC side, we implement a complete interrupt and exception system to handle relevant events. This system supports the sending/receiving of interrupts to/from other control cores or the GP zone. On the GP side, `hthreads` supports the sending/receiving of interrupts to/from all the control cores.

There are two ways to implement the interrupt mechanism: (1) to simulate it by busy waiting, and (2) to implement it based on the hardware features of MT-3000. The former does not require hardware support but consumes computing resources, whereas the latter does not consume computing resources but requires hardware support. We implement both approaches, and compare their performance.

To reduce the synchronization cost between GP and ACC zones, we design an optimized synchronization model. Fig. 7a is the direct synchronization model, and Fig. 7b is the proxy synchronization model. In the direct synchronization model, the host thread in the GP zone needs to synchronize with all the ACC cores. In the proxy synchronization model, the proxy core synchronizes with other cores, and then it synchronizes with the host thread in the GP zone. The synchronization overhead between the ACC cores is smaller than that between the GP and ACC zones. So, the proxy synchronization model performs better than the direct synchronization model, and consumes fewer GP zone resources.

4.2.3 ACC zone runtime

To minimize the kernel launching overhead, we implement an OS-like runtime for the ACC zone.

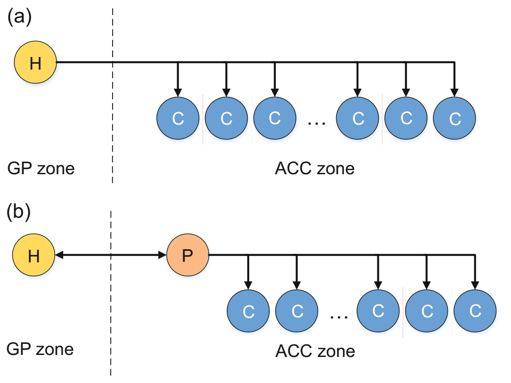


Fig. 7 The hthreads communication model: (a) direct model (the host thread in the GP zone communicates with all ACC cores directly); (b) proxy model (the host thread in the GP zone communicates only with the proxy core in the ACC zone, which then communicates with other ACC cores). GP: general-purpose; ACC: acceleration; H: host; P: proxy core; C: ACC core

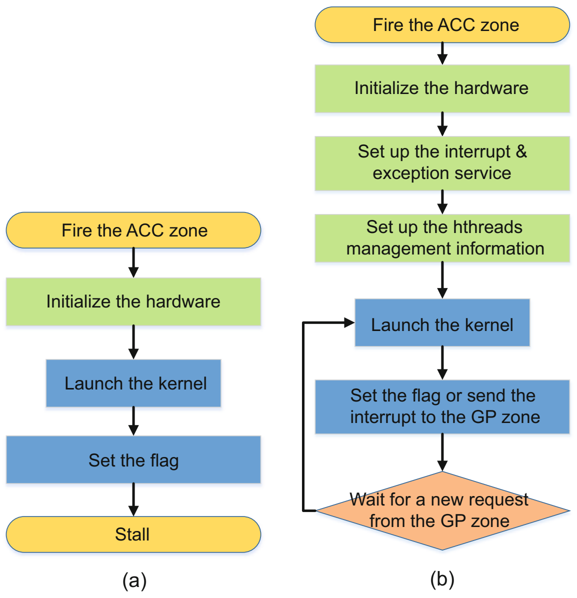

Fire the ACC zone
Initialize the hardware
Launch the kernel
Set the flag
Stall
(a)
Fire the ACC zone
Initialize the hardware
Set up the interrupt & exception service
Set up the hthreads management information
Launch the kernel
Set the flag or send the interrupt to the GP zone
Wait for a new request from the GP zone
(b)

Fig. 8 The ACC zone runtime: (a) naive ACC runtime (which will stall after the user task completion); (b) OS-like ACC runtime (which will still run on the ACC zone after the user task completion, and wait for new tasks). ACC: acceleration; OS: operating system; GP: general-purpose

Fig. 8a shows the naive ACC zone runtime, which supports kernel execution, but will stall after kernel completion. With this naive runtime, users have to fire the ACC zone for each kernel execution, which suffers from a large overhead. To avoid repeatedly switching the ACC zone on/off, we implement an OS-like runtime for the ACC zone, as shown in Fig. 8b. We see that it not only supports kernel execution, but sets up the interrupt and exception services that can report kernel runtime errors. In addition, the OS-like runtime will still run on the ACC zone after kernel completion, and wait for a new request from the GP zone. A new request can be a new kernel execution or other operations such as synchronization. A detailed evaluation of the kernel launching overhead will be shown in Section 6.2.2. We will see that the OS-like runtime can yield better performance than the naive one.

4.2.4 Vector extension

To orchestrate the use of the 16 acceleration cores of an acceleration array, we extend standard C to support vector data types and vector operations. The use of vector data types is demonstrated in lines 19–21 of Fig. 5, where the qualifier `lvector` is followed by the relevant conventional data types.

These vector extensions are designed to give the programmer the ability to explicitly control the 16 acceleration cores working in a lock-step manner. They can also be used by the `m3cc` compiler to exploit the potential of the acceleration array in the ACC zones of MT-3000.

4.2.5 Debugging supports

As MT-3000 is a new heterogeneous many-core accelerator, and its ACC zone is a bare-metal device, programming the ACC zone is error-prone and time-consuming. The debugging process requires a deep understanding of the hardware architecture. To help users debug, we provide a `printf` function on the ACC side. When there occurs an exception from device kernels, `hthreads` will print the kernel call stack on the GP side. We also implement a *gdb-like* tool (i.e., `et_ctl`) to help users debug their codes. It supports common debugging functions, such as setting breakpoints, showing the contents of registers or memory, and step execution. We are currently enriching the tool to enable it to capture more information.

5 MOCL3

`MOCL3` is an implementation of the OpenCL standard parallel programming interface for the MT-3000 architecture. It follows the programming specification of OpenCL (version 1.2). In general, the implementation of the OpenCL programming model for MT-3000 includes two parts: the kernel compiler and the runtime system. The kernel compiler compiles OpenCL kernels into MT-3000 binaries, and the runtime system implements the programming interfaces defined by the OpenCL specification. `MOCL3` is an upgraded version of the OpenCL programming system that was developed originally for Matrix-2000, known as the OpenCL programming interface for Matrix-2000 (`MOCL`) (Jääskeläinen et al., 2015; Zhang et al., 2018).

5.1 OpenCL kernel compiler

From the programmer's point of view, an OpenCL program includes two parts: host side code and device side code (i.e., *kernels*). When compiling a kernel, we need to compile the OpenCL C code into MT-3000 binaries. The OpenCL kernel is written according to the OpenCL C (based on C99) specification, but it also has syntax extensions and constraints.

Fig. 9a shows that our kernel compiler for MT-3000 is implemented in three steps. We first convert OpenCL kernels into workgroup functions with a loop (WGF), represented in low-level virtual machine (LLVM) intermediate representation (IR). According to the index space defined by the OpenCL program, the translated program is the task to be performed by a single workgroup function. Different workgroup functions share the same code, but access different data elements

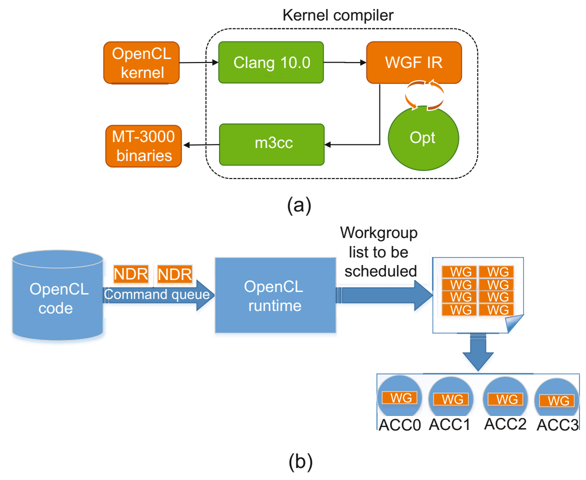


Fig. 9 Implementation of MOCL3: (a) MOCL3 kernel compiler; (b) MOCL3 runtime. WGF: workgroup function; IR: intermediate representation; Opt: optimizer; NDR: NDRange; WG: workgroup; ACC: acceleration

through the index space. Second, we perform optimizations on the WGF IR codes with a customized optimizer. Due to the lack of an LLVM backend for MT-3000, we use the `llvm-cbe` tool (https://github.com/JuliaComputing/llvm-cbe/) to translate the WGF IRs into C codes. Third, we compile the workgroup functions into MT-3000 binary representations with the `m3cc` compiler.

1. Handling local variables

The local variables of OpenCL kernels are shared among all the work-items of the same workgroup. We convert such local variables to an additional workgroup function argument with a fixed allocation size. During runtime, the local variables are mapped to a predefined address of the on-chip buffers, i.e., either SM or AM (Fig. 2). Given that programmers have to manually move data between on-chip buffers and off-chip DDR memories, we implement the relevant builtins (`async_work_group_copy`) to assist data movements between local and global memories.

2. Atomics implementations

OpenCL C provides atomic operations to locations in `__global` or `__local` memory. In `MOCL3`, a workgroup is translated into a work-item loop by our kernel compiler, which is scheduled to a hardware thread. The work-items within a workgroup are executed one by one and in a sequential fashion. In terms of memory access, the work-items of this workgroup will access local variables sequentially. Therefore, the atomic operations on local memories can be replaced by equivalent functional operations without synchronization. For the global memory case, we reply on the hardware locks of MT-3000 to implement the atomic functions. Again, `hthreads` provides relevant APIs to manage the shared resource such as *locks*.

5.2 MOCL3 runtime system

When we implement the OpenCL runtime system on MT-3000, the key is to implement OpenCL APIs. OpenCL programming interfaces are used mainly to manage the interaction between the host and the accelerator, including creating a context environment, managing the program object and compilation at the ACC zone, managing the buffers and data movements between the host zone and the ACC zone, and starting the kernel program at the ACC zone. Our OpenCL runtime system on MT-3000 is built on the heterogeneous driver and `hthreads` (Fig. 3).

According to the aforementioned OpenCL kernel compilation process, a large number of concurrent tasks (i.e., workgroup functions) are defined by the index space. After the kernel program is started at the accelerated zone, the OpenCL runtime system needs to dispatch tasks during runtime. The process of executing an OpenCL program during runtime is shown in Fig. 9b. Taking the OpenCL application as the input, the kernel (as well as the corresponding NDRange) is submitted to the computing device for execution through the OpenCL task queue, which is managed by the runtime system; on the device side, a workgroup is used as the basic unit for assigning tasks to available acceleration array cores. Considering that the OpenCL program adheres to a parallel model of data structure, the task mapping strategy used is a static one.

6 Results

This section evaluates how `hthreads` and `MOCL3` perform on MT-3000, and compares the design tradeoffs when implementing the programming interfaces for MT-3000.

6.1 Conformance test

Since `hthreads` is a new heterogeneous programming interface specifically targeting MT-3000, we have developed a large number of internal test cases, also known as the `ht-bench` suite, which aims to cover all the APIs of `hthreads`. Up to now, our `hthreads` library can successfully verify all the test cases.

As for `MOCL3`, we use the test cases built in the POCL repository (http://portablecl.org/). Our results demonstrate that `MOCL3` can pass all the 125 test cases, covering those of kernel compilation, runtime, workgroups, and regression. This shows that `MOCL3` respects the OpenCL programming specification.

6.2 Design tradeoffs

6.2.1 Memory and code layout

1. Memory layout

The placement of stack has a significant impact on performance. As for MT-3000, the stack can be placed on DDR, GSM, or SM. We have compared their performance using a micro-benchmark (i.e., a variant of vector addition) in `hthreads`. Fig. 10a shows the execution time of the three policies, i.e., corresponding to placing the stack on SM, GSM, or DDR. We see that placing the stack on SM yields the best performance, with an average performance improvement of 14% over DDR. Given that the micro-benchmark is simple, we believe that this placement policy can achieve a larger performance speedup for large real-life codes.

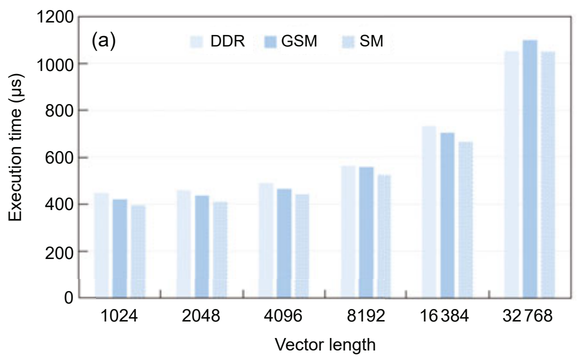
On the other hand, using the SM buffer is often beneficial for applications' performance by exploiting data locality. Thus, we should use this on-chip buffer carefully, i.e., using it as a stack or data cache. To this end, we provide programmers with an environment variable `HT_MEM_LAYOUT_POS`={0, 0.2, 0.4, 0.6, 0.8, 1.0} in the production environment. When it equals 0.2, we use 20% of SM per core as the stack space, and programmers can use the remaining 80% space. When `HT_MEM_LAYOUT_POS`=1.0, we choose to use 100% of SM per core as the stack space, and programmers have no access to this buffer. In this way, programmers can fully use SM buffers according to their applications.

2. Code layout

We evaluate the impact, in terms of performance, of choice of the location in which the code segments are stored. In MT-3000, we can map the code segments on either DDR or GSM. Fig. 10b shows the improvement obtained in terms of performance as a result of placing code on GSM rather than DDR. We see that the incremental performance benefit obtained is within 5%. This is because the acceleration core of MT-3000 has an instruction cache, and the location of code segments has little impact on the overall performance. Note that the GSM buffer can also be used as the on-chip data buffer to exploit data locality. Therefore, we provide programmers with an environment variable `HT_CODE_LAYOUT_POS`={0, 1}. When `HT_CODE_LAYOUT_POS`=0, the code segments of `hthreads` kernels are mapped onto the DDR space; otherwise, they are mapped onto the GSM space.

6.2.2 Launching overhead

One of the design goals of `hthreads` is to minimize its management overhead, and hereby we evaluate the kernel launching overhead. There are two ways to launch a kernel: one is to fire the



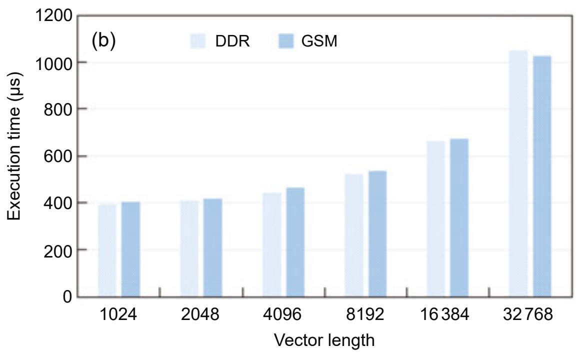


Fig. 10 Performance comparison of different design choices: (a) memory layout (the performance obtained from placing stack on SM over DDR and GSM); (b) code layout (performance comparison between placing code on DDR and GSM). SM: scalar memory; DDR: double data rate; GSM: global shared memory

acceleration cores directly, and the other is to use interrupt to wake them up. Accordingly, there are two ways to check kernel completion: one is to use query flags, and the other is to use interrupts. In `libMT`, users can start the kernel only by firing the acceleration cores one by one, and then query the flags of each core. In `hthreads`, users can start the kernel by firing the acceleration cores with the `hthread_group_create` API, or start it using interrupt with the `hthread_group_exec` API. Checking kernel completion in `hthreads` is implemented by interrupts.

Since interrupts can either be implemented by hardware or be simulated by querying flags, we have evaluated the performance of both approaches. Fig. 11 shows the launching overheads of using hardware interrupt over the querying approach. We see that the overhead can be reduced by up to 50% by starting the kernel with interrupt compared with firing cores one by one. The interrupt mode can be supported only by our OS-like runtime. Using query flags to simulate interrupts increases the

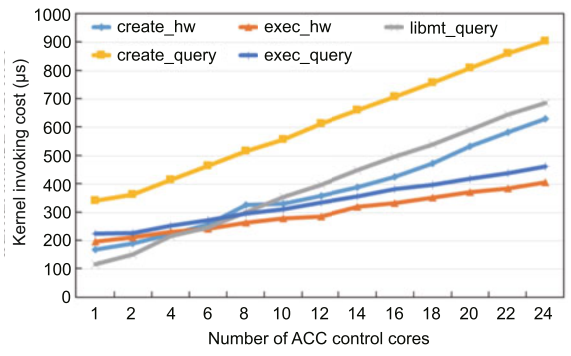


Fig. 11 Invoking cost of using hardware interrupt over querying

overhead by 14% compared to using hardware interrupts. Compared to `libMT`, we can lower the launching overhead of using 24 cores from 700 to 400 μs. When porting applications to Matrix-3000, programmers need to make sure that the kernel execution time is much larger than the launching overhead; otherwise, they cannot obtain any performance benefit. They should fire the ACC zone once with the `hthread_group_create` API, and use the `hthread_group_exec` API to launch their kernel to minimize the launching overhead.

6.3 Performance optimization

General matrix multiplication (GEMM) is a fundamental building block for HPC applications, from traditional scientific simulations to emerging deep learning workloads. GEMM is a matrix-multiply-accumulate operation, defined as $\boldsymbol{C} = \alpha\boldsymbol{A}\cdot\boldsymbol{B} + \beta\boldsymbol{C}$, where $\boldsymbol{A}$ and $\boldsymbol{B}$ are matrix inputs, α and β are scalar inputs, and $\boldsymbol{C}$ is a pre-existing matrix that is overwritten by the output. Here, matrix $\boldsymbol{A} \in \mathbb{R}^{M\times K}$, matrix $\boldsymbol{B} \in \mathbb{R}^{K\times N}$, and $\boldsymbol{C} \in \mathbb{R}^{M\times N}$. As a case study, we have implemented matrix multiplications in `hthreads` and `MOCL3`.

6.3.1 Optimizing GEMM with `hthreads`

As a case study, we have implemented and optimized matrix multiplications with the `hthreads` APIs. The performance speedups of using various optimizations are shown in Fig. 12a. We take the naive parallel implementation with `hthreads` as the baseline. Based on this, we have performed mainly loop tiling (1D or 2D) and used on-chip memory (SM or AM). 1D represents tiling the loop on one dimension, whereas 2D represents tiling the loop on two dimensions. To achieve data locality, we stage the tiled

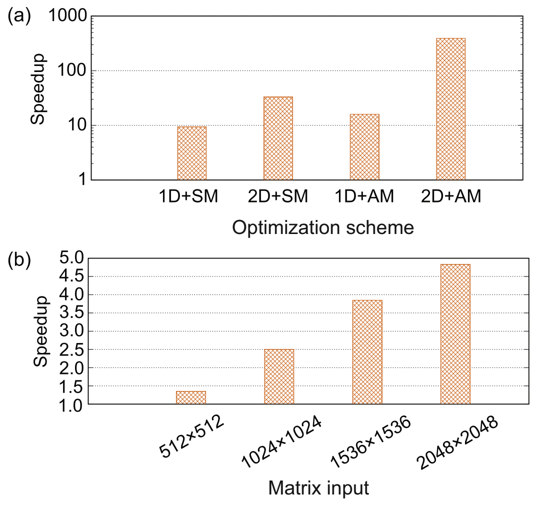


Fig. 12 Speedup over baseline implementations for matrix multiplications: (a) with various optimizations in hthreads; (b) with local memory in MOCL3

data on SM or AM. The usage of on-chip buffers is achieved by exploiting `hthreads` DMA APIs to move data between off-chip memory and on-chip memory. On MT-3000, we see that the achieved speedups of the four optimizations are 9.4×, 33.04×, 15.98×, and 392.29×.

6.3.2 Optimizing GEMM with `MOCL3`

We have also implemented matrix multiplications with `MOCL3`. Fig. 12b shows the performance improvement of using local memory over the case without local memory on MT-3000 for various inputs ($M=N=K=\{512, 1024, 1536, 2048\}$). We see that by mapping the local memory to the on-chip memory of MT-3000, `MOCL3` can run matrix multiplications around 2.8× faster on average. By moving data into the on-chip memories of MT-3000, we can improve the memory bandwidth by exploiting the fast on-chip buffers and reusing the data elements. Note that we use the `async_work_group_copy` *builtins* to move data from global memories to local memories.

To summarize, we have used matrix multiplications as a case study to demonstrate the performance of `hthreads` and `MOCL3`. We find that the programming model developed for MT-3000 performs well in terms of both performance and programmability.

7 Related works

The parallel programming model acts as a bridge between programmers and parallel architectures. To use shared memory parallelism on multicore CPUs, parallel programming models are often implemented on threading mechanisms such as the POSIX threads (Alfieri, 1994). For programming heterogeneous many-core processors, Fang et al. (2020) summarized the family of parallel programming models for heterogeneous many-core architectures. Based on the performance–programmability tradeoff, the programming models/languages were categorized into low- and high-level programming models. The expected application performance increases from high- to low-level programming models, whereas the programmability decreases.

Low-level programming models are closer to many-core architectures, and expose more hardware details to programmers through data structures and/or APIs. These models are typically bound to specific hardware architectures, and are also known as native programming models. The representative models are libSPE for STI Cell/B.E. (Arevalo et al., 2000) and CUDA for NVIDIA GPUs (https://developer.nvidia.com/cuda-downloads). In contrast, high-level programming models raise the languages' abstraction level, and hide more architecture details than low-level models. Thus, high-level models often enable better programmability. The representative models are SYCL (https://www.khronos.org/sycl/), Kokkos (Trott et al., 2022), OpenACC (https://www.openacc.org/), and PyTorch (https://pytorch.org/).

To achieve high performance, high programmability, and high portability, we argue that a holistic solution of programming systems is required for future heterogeneous many-core processors. In this paper, we share our experience on the development of programming systems for our home-grown heterogeneous processor. At the low level, we present a close-to-metal programming interface (`hthreads`) to tap the hardware potentials. At the high level, we present a customized implementation of the standard programming interface (`MOCL3`). This holistic solution aims to achieve a balance among performance, programmability, and portability.

8 Conclusions

We have presented the design and implementation of the programming and compiler tools for the Matrix-3000 accelerator. Given the complex memory hierarchy and processor core organization of Matrix-3000, the use of microarchitecture design in a way that enables complete realization of potential hardware performance depends on the effectiveness of the customized software employed on the device. We share our experience on how a low-level threading-based programming interface can be developed to support the high-level OpenCL programming standard. We hope that the experience shared in this paper can support the design and implementation of system software for future specialized computing hardware.

Contributors

Chun HUANG, Kai LU, and Ruibo WANG designed the research. Jianbin FANG and Peng ZHANG processed the data. Chun HUANG, Tao TANG, and Zheng WANG drafted the paper. Jianbin FANG helped organize the paper. All the authors revised and finalized the paper.

Compliance with ethics guidelines

Jianbin FANG, Peng ZHANG, Chun HUANG, Tao TANG, Kai LU, Ruibo WANG, and Zheng WANG declare that they have no conflict of interest.

Data availability

The data that support the findings of this study are available from the corresponding author upon reasonable request.

References

Alfieri RA, 1994. An efficient kernel-based implementation of POSIX threads. Proc USENIX Summer Technical Conf, p.59-72.

Arevalo A, Matinata RM, Pandian M, et al., 2000. Programming the cell broadband engine examples and best practices. ACM Workshop. Available from https://www.autodesk.com/research/publications/programming-the-cell-broadband [Accessed on Aug. 25, 2022].

Fang JB, Varbanescu AL, Sips H, 2011. A comprehensive performance comparison of CUDA and OpenCL. Int Conf on Parallel Processing, p.216-225. https://doi.org/10.1109/ICPP.2011.45

Fang JB, Huang C, Tang T, et al., 2020. Parallel programming models for heterogeneous many-cores: a comprehensive survey. *CCF Trans High Perform Comput*, 2(4):382-400. https://doi.org/10.1007/s42514-020-00039-4

Jääskeläinen P, de la Lama CS, Schnetter E, et al., 2015. pocl: a performance-portable OpenCL implementation. *Int J Parall Program*, 43(5):752-785. https://doi.org/10.1007/s10766-014-0320-y

Kudlur M, Mahlke S, 2008. Orchestrating the execution of stream programs on multicore platforms. Proc 29th ACM SIGPLAN Conf on Programming Language Design and Implementation, p.114-124. https://doi.org/10.1145/1375581.1375596

Liao XK, Lu K, Yang CQ, et al., 2018. Moving from exascale to zettascale computing: challenges and techniques. *Front Inform Technol Electron Eng*, 19(10):1236-1244. https://doi.org/10.1631/FITEE.1800494

Lu K, Wang YH, Guo Y, et al., 2022. MT-3000: a heterogeneous multi-zone processor for HPC. *CCF Trans High Perform Comput*, 4(2):150-164. https://doi.org/10.1007/s42514-022-00095-y

Owens JD, Luebke D, Govindaraju N, et al., 2005. A survey of general-purpose computation on graphics hardware. Proc 26th Annual Conf of the European Association for Computer Graphics, p.21-51. https://doi.org/10.2312/egst.20051043

Owens JD, Houston M, Luebke D, et al., 2008. GPU computing. *Proc IEEE*, 96(5):879-899. https://doi.org/10.1109/JPROC.2008.917757

Patterson D, 2018. 50 years of computer architecture: from the mainframe CPU to the domain-specific TPU and the open RISC-V instruction set. IEEE Int Solid-State Circuits Conf, p.27-31. https://doi.org/10.1109/ISSCC.2018.8310168

Perez JM, Bellens P, Badia RM, et al., 2007. CellSs: making it easier to program the cell broadband engine processor. *IBM J Res Dev*, 51(5):593-604. https://doi.org/10.1147/rd.515.0593

Shen J, Fang JB, Sips H, et al., 2012. Performance gaps between OpenMP and OpenCL for multi-core CPUs. Proc 41st Int Conf on Parallel Processing Workshops, p.116-125. https://doi.org/10.1109/ICPPW.2012.18

Trott CR, Lebrun-Grandié D, Arndt D, et al., 2022. Kokkos 3: programming model extensions for the exascale era. *IEEE Trans Parall Distrib Syst*, 33(4):805-817. https://doi.org/10.1109/TPDS.2021.3097283

Zhai JD, Chen WG, 2018. A vision of post-exascale programming. *Front Inform Technol Electron Eng*, 19(10):1261-1266. https://doi.org/10.1631/FITEE.1800442

Zhang P, Tang T, Fang J, et al., 2018. MOCL: an efficient OpenCL implementation for the Matrix-2000 architecture. Proc 15th ACM Int Conf on Computing Frontiers, p.26-35. https://doi.org/10.1145/3203217.3203244