Hindawi Publishing Corporation
Mobile Information Systems
Volume 2016, Article ID 4513486, 24 pages
http://dx.doi.org/10.1155/2016/4513486

Research Article

An OpenMP Programming Environment on Mobile Devices

Tyng-Yeu Liang, Hung-Fu Li, and Yu-Chih Chen

Department of Electrical Engineering, National Kaohsiung University of Applied Sciences, No. 415, Jiangong Road, Sanmin District, Kaohsiung City 807, Taiwan

Correspondence should be addressed to Tyng-Yeu Liang; lty@mail.ee.kuas.edu.tw

Received 16 January 2016; Revised 22 April 2016; Accepted 22 May 2016

Academic Editor: Alessandro Cilardo

Recently, the computational speed and battery capability of mobile devices were greatly promoted. With an enormous number of APPs, users can do many things in mobile devices as well as in computers. Consequently, more and more scientific researchers are encouraged to move their working environment from computers to mobile devices for increasing their work efficiency because they can analyze data and make decisions on their mobile devices anytime and anywhere. Accordingly, we propose a mobile OpenMP programming environment called MOMP in this paper. Using this APP, users can directly write, compile, and execute OpenMP programs on their Android-based mobile devices to exploit embedded CPU and GPU for resolving their problems without network connection. Because of source compatibility, MOMP makes users easily port their OpenMP programs from computers to mobile devices without any modification. Moreover, MOMP provides users with an easy interface to choose CPU or GPU for executing different parallel regions in the same program based on the properties of parallel regions. Therefore, MOMP can effectively reduce the programming complexity of heterogeneous computing in mobile devices and exploit the computational power of mobile devices for the performance of user applications.

1. Introduction

In recent years, the hardware of mobile devices such as smartphones and tablets has been obviously upgraded. Most of modern mobile devices have multicore CPU like ARM, many-core GPU such as Adreno, PowerVR, Krait, and NVIDIA Tegra, 1~2 GB RAM, and enough battery capability to standby for dozens of hours. This remarkable hardware advancement has made mobile devices comparable to common PCs in the execution performance of applications. On the other hand, the amount and type of applications appearing in Google Play and APP store are enormous and diverse. A lot of software programs such as MS Office, Skype, Adobe Professional, and Photoshop have released mobile versions. With these APPs, users can do many things in mobile devices as well as in PCs. Consequently, mobile devices have become the most important equipment for users to connect with networks for handling their daily affairs including communication, working, shopping, learning, and playing because they are easier than laptops to be owned and carried.

Reacting to this development trend, some scientific software such as Octave [1] and Addi [2] recently appeared in Google Play to provide users with MATLAB-like environment. Using these two APPs, researchers and engineers can make use of CPUs embedded in mobile devices to perform mathematic computation and simulation by means of MATLAB [3] instructions or scripts. As a result, they can effectively increase their work efficiency because they usually carry their smartphones with themselves anytime anywhere and can continuously do their researches on the smartphones even when they leave from offices or laboratories. Moreover, they can save energy consumed for resolving their problems since the processors embedded within mobile devices usually are higher in energy efficiency than the processors of computers.

However, not all of problems are resolvable only by using scientific simulation software. When users intend to exploit the computation power of multicore CPUs or many-core GPUs for speeding up data computation, they have to write multiprocess or multithreading programs by using MPI [4], Pthread, OpenMP [5], CUDA [6], or OpenCL [7]. Unfortunately, most of the programs developed by these programming APIs are not portable onto mobile devices because mobile devices are different from computers in processor

architecture and operating system and do not support necessary toolkits and runtime libraries. Although Java is effective for resolving the problem of resource heterogeneity, it is not as efficient as C/C++. Therefore, user programs usually have to consume more time and energy if they are developed by Java instead of C/C++. For resolving this problem, several APPs such as C4droid [8], CppDroid [9], and CCTools [10] recently were proposed for users to develop and execute C/C++ programs on mobile devices. Basically, these APPs provide an embedded Linux terminal for users to develop programs. Consequently, they allow users to write multithreaded programs by Pthread for exploiting multicore CPU in mobile devices. However, they have some problems as follows.

First, the user interface of Linux terminal is not friendly for those who are not used to operating Linux OS. Second, multithreading programming is not easy enough for users. When users make use of Pthread to develop applications, they have to manually deal with problem partition, data and thread synchronization, and load balance by themselves. Consequently, the programming effort of users is increased along with the code length of user programs. This is a big burden for users to write programs on mobile devices especially when the size of touch screen is not big enough. Third, these APPs do not support GPU programming in mobile devices. Although some smartphones support CUDA/OpenCL runtime library and driver, no APP currently can provide an IDE for users to develop CUDA or OpenCL applications on mobile devices. They must write CUDA/OpenCL programs in computers and download the programs from computers to mobile devices for execution. As a result, it is not convenient for users to exploit the computation power of GPUs in mobile devices for resolving their problems especially at the phase of application development because they need to repeat remote program modification and recompilation many times through unstable wireless networks.

As previously described, we propose a mobile OpenMP programming environment called MOMP in this paper. Using this APP, users can directly edit, compile, and execute OpenMP programs on mobile devices and exploit the computational power of embedded CPU and GPU for resolving their problems. OpenMP is a directive-oriented API. When users develop parallel applications with OpenMP, what they need to do is to add OpenMP directives into their C/C++ programs. The OpenMP compiler of MOMP can generate Pthread and OpenCL codes based on the semantic of OpenMP directives and can automatically add the instructions necessary for data partition and synchronization into the generated programs. As a result, MOMP can effectively reduce the programming complexity of heterogeneous computing in mobile devices because it allows users to develop parallel applications for exploiting the computational power of embedded CPU and GPU with a uniform and directive-oriented programming interface, that is, OpenMP, which is much easier than Pthread, CUDA, and OpenCL. Moreover, MOMP provides users with an extended directive to choose CPU or GPU for executing different parallel regions in the same program according to computation demand and parallelism in order to obtain the best program performance.

The rest of this paper is organized as follows. Section 2 is the background of OpenMP and OpenCL. Sections 3 and 4 describe the framework and implementation of MOMP, respectively. Section 5 discusses the experimental results of performance evaluation. Section 6 is the discussion of related work. Finally, Section 7 gives a number of conclusions for this paper and our future work.

2. Background

OpenMP is a shared memory parallel programming interface managed by OpenMP architecture review board. It consists of compiler directives, runtime library, and environment variables. Users can easily control the parallelism of their C/C++/Fortran programs by using OpenMP directives. The parallelism of OpenMP basically is classified into two kinds, that is, task parallelism and data parallelism. Task parallelism is to create threads for executing different sections while data parallelism is to generate threads for sharing the work of the same for-loop. On the other hand, users can make use of scheduling-clauses such as *static*, *dynamic*, and *guide* or the OpenMP runtime functions to determine the way of work sharing in a given parallel region. For thread synchronization, they can use *atomic* to specify which variable should be updated atomically. They also can use data clauses to control the attribute of shared variables. For example, the *private* clause makes the threads of the same parallel region to have their own copy of the same variable. The *firstprivate* clause makes these copies be initialized with the original value of the variable. The *lastprivate* clause makes the variable be updated with the value of the copy in the last thread that leaves from the parallel region. The *reduction* clause can make the values of the copies be reduced by a specific operation such as add, subtract, multiple, divide, max, and min. Because the OpenMP compiler can automatically generate the corresponding multithreading codes according to OpenMP directives, OpenMP is effective for reducing the complexity of multithreading programming, and thereby it has become a popular shared memory parallel programming standard for shared memory multiprocessors. Many researches [11, 12] were aimed at enabling OpenMP programming interface based on software distributed shared memory systems [13, 14] for reducing the programming complexity of computational clusters. In recent years, GPGPU has successfully become an alternative for high performance computing because it can provide high computation performance with low energy. Because the complexity of GPU programming such as CUDA and OpenCL is too high for most of users, many OpenMP-to-CUDA or OpenCL compilers such as Cetus [15], OpenMPC [16], OMPi [17], and HMCSOT [18] were proposed to address this issue.

OpenCL is an open programming interface proposed by Khronos Group to support heterogeneous computing. The runtime library of OpenCL consists of Platform, Device, Context, Program, Kernel, MemoryObject, CommandQueue, Image, and Sampler. The execution flow of OpenCL programs basically is composed of the steps as follows. The first is to get a platform available in the execution node and allocate an available device in the platform by clGetPlatformIDs() and

clGetDeviceIDs(). The second is to build a context executable on the allocated device and construct a CommandQueue for the context to accept the command from the user program by clCreateContext() and clCreateCommandQueue(). The third is to generate a source program and compile the source program into an executable program by clCreateProgramWithSource() and clBuildProgram(), respectively. The fourth is to create a kernel for obtaining the entry point of the kernel function in the executable program by clCreateKernel(). The fifth is to allocate input and output MemoryObjects (i.e., device memory) accessible for the kernel and then copy data from host memory to the input MemoryObjects by clCreateBuffer() and clEnqueueWriteBuffer(), respectively. The sixth step is to launch the kernel function to the allocated device for execution by clEnqueueNDRangeKernel(). The seventh is to copy the execution result from the output MemoryObjects to host memory by clEnqueueReadBuffer() after the execution of the kernel function finishes. Recently, several studies such as JCL [19], VCL [20], and SunCL [21] were dedicated to the implementation of OpenCL cluster for resolving problems with a cluster of distributed heterogeneous processors.

Different from the previous work, this work is aimed at proposing a mobile programming environment for reducing the programming complexity of heterogeneous computing on mobile devices. In the proposed programming environment, OpenMP programs are translated into OpenCL ones which can be executed by CPU or GPU. For the applications implemented with recursive algorithms, dynamic task creation and task dependency maintenance are necessary. However, OpenCL does not support dynamic parallelism until the second version. Almost all mobile devices produced by different vendors currently support only OpenCL 1.1 or 1.2 embedded profile, which does not provide dynamic parallelism. Therefore, this work currently is focused on the implementation of OpenMP 2.0 on mobile devices for data-parallelism applications.

3. Framework

MOMP is developed based on a mobile integrated development environment called UbiC, which was proposed by our previous work [22]. With the support of UbiC, users can edit, compile, execute, and debug their C programs on Android-based mobile devices without the help of remote servers. UbiC consists of GUI, program editor, compiler, executer, and debugger as shown in Figure 1. UbiC adopts Clang [23] to compile C programs into LLVM [24] IRs. When users press the execution button, the executor of UbiC loads the LLVM IRs of the working program and translates LLVM IRs into optimized native codes for execution by means of MCJIT. Consequently, UbiC can allow user programs to be portable onto any platform that supports LLVM while simultaneously obtaining a good performance as well as native codes. On the other hand, UbiC adopts a modified LLVM interpreter instead of gdb to be program debugger because of cost consideration. It inserts DWARF-3 tags into user programs when it compiles the programs. When users debug their

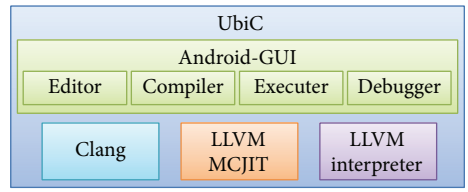

Figure 1: Framework of UbiC.

programs, the modified LLVM interpreter retrieves the necessary information from program contexts for users to debug their programs according to the inserted tags. Currently, the program debugger supports *breakpoint*, *single step*, and *variable view* for users to trace the execution status of their programs. Although UbiC provides a complete and friendly C programming environment, it does not support OpenMP. Basically, the GUI interface of MOMP is as same as that of UbiC while the compiler and executor of MOMP are different from those of UbiC because MOMP is dedicated to supporting OpenMP. Therefore, the following description is focused on the compiler and executor of MOMP.

The program compiler of MOMP mainly consists of Java-based OmniDriver, frontend and backend compiler. The frontend compiler is constructed by yacc and C while the backend compiler is implemented by Java. Here we give an example program to explain the flow of program compilation in MOMP as shown in Figure 2. This example program is composed of two parallel regions: one is matrix multiplication and another is vector addition. When users press the compilation button of GUI to compile this program, the program compiler of MOMP will compile the program as follows.

First, the compiler driver sets up the arguments and path of program compilation and expands the source program by the preprocessor of Clang. Second, the frontend compiler translates the source program into an abstract syntax tree which are composed of X objects. Third, the backend compiler generates another source program based on the abstract syntax tree. In the new source program, the two parallel regions are extracted from the main function and then translated into Pthread modules (e.g., _ompc_func_1 and _ompc_func_2) and OpenCL ones (e.g., _cuker_1 and _cuker_2). On the contrary, the main functions are modified by replacing the two parallel regions with the stub1 and stub2 function, which are used to invoke the Pthread-version modules or OpenCL-version modules according to the architecture of target processor. Finally, Clang compiles the new source program, which consists of main module, Pthread modules, and OpenCL modules to generate an executable file (i.e., output.ll) of LLVM IRs. The OpenMP translator of the backend compiler is implemented based on OMPICUDA, which was proposed by our previous work [25]. The implementation of the OpenMP translator will be detailed in Section 4.

When users press the execution button of GUI to run the example program, the executor of MOMP loads the executable file, that is, output.ll, of the program, and then executes the executable file through LLVM by means of MCJIT,

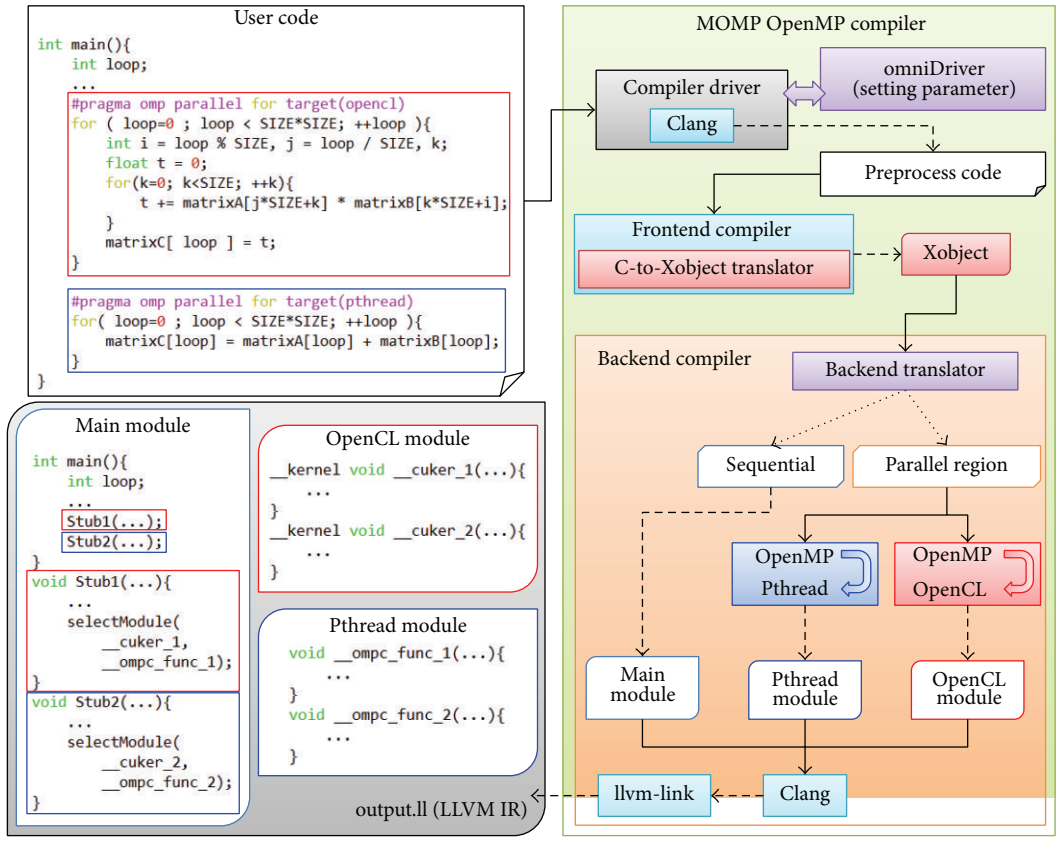

Figure 2: Flow of program compilation in MOMP.

as shown in Figure 3. Because the executable file is runnable on any platform that supports LLVM, MOMP effectively maintains the portability of user programs. It is worth saying that MOMP provides an extended directive called target() for users to select the Pthread or OpenCL version for a given parallel region to be executed on runtime based on parallelism degree and computation demand. If the target directive is set as "pthread," the executor loads the Pthread-version module of the parallel region and executes the module by CPU. By contrast, it loads the OpenCL-version module and executes the module with GPU by default if the target directive is set as "opencl." However, not all of mobile devices support both of GPU and the OpenCL driver. For example, MiPAD supports only the CUDA driver for its NVIDIA K1 GPU. Consequently, the executor loads the OpenCL-version module and executes the module with GPU if both of the OpenCL driver and GPU are available. Otherwise, it loads the OpenCL-version module while executing the module with other types of processors like ARM CPU if the OpenCL driver but GPU is available. If both of OpenCL and GPU are not available, it automatically changes to load the Pthread-version module of the parallel region and execute the module by ARM CPU. By means of the target directive, users can easily adapt the type of processors used to execute different parallel regions for optimizing the total execution performance of their programs through the same programming interface.

4. Implementation

Basically speaking, there are two important jobs in the implementation of MOMP. The first is to port the OpenMP compiler of OMPICUDA onto mobile devices. The second is to implement a reduced OpenCL runtime library based on CUDA driver API for NVIDIA mobile GPU such as K1. We detail how to accomplish these two jobs as follows.

4.1. OpenMP Compiler. The entry point of the OpenMP compiler consists of a launcher and omniDriver. They were originally implemented by C and shell script, respectively. The launcher is responsible of passing the arguments of program compilation to omniDriver for creating a template. This template stores the necessary information such as native compiler, programming language, and the paths of runtime and dynamic linking libraries such as "-L/libraryPATH/...-lpthread" for the backend of the OpenMP compiler to analyze and translate user programs into executable files. To accomplish the functions of launcher and omniDriver, we faced and resolved the problems on Android for MOMP as follows.

Although Android is constructed based on Linux, it does not support bash or all the shell instructions. Unfortunately, this problem results in that the omniDriver implemented by shell script is not workable in Android. For resolving this problem, we implemented a new launcher and omniDriver

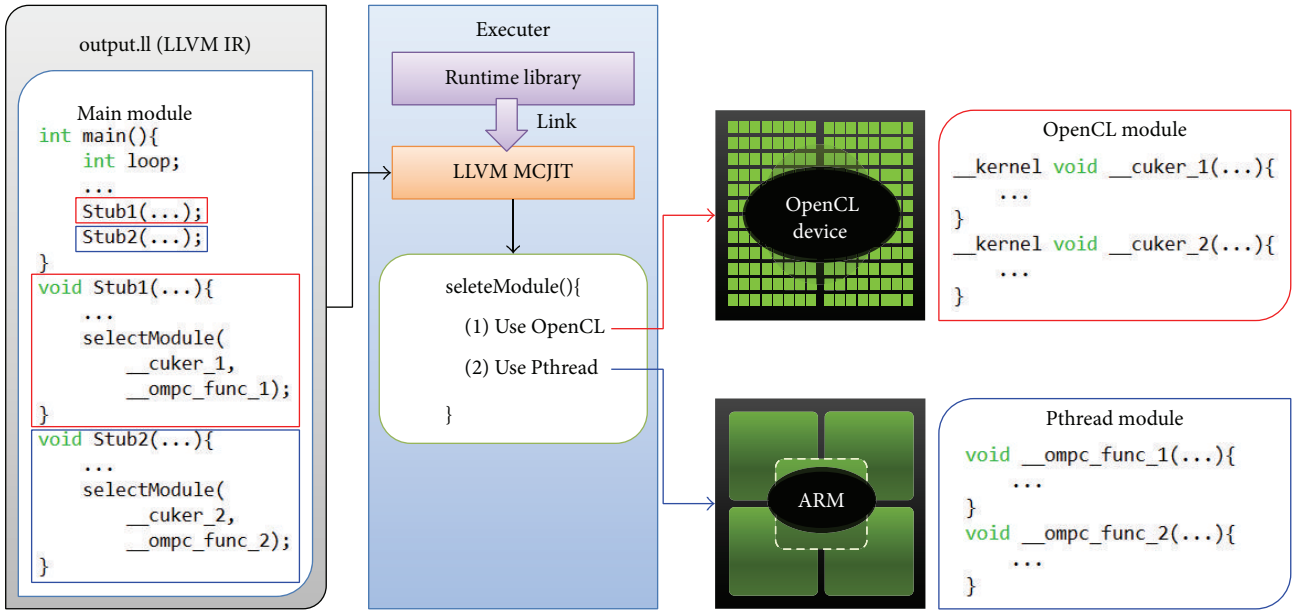

Figure 3: Program execution in MOMP.

by means of Java programming language in order to connect with the user interface and the backend of the OpenMP compiler in MOMP. On the other hand, most of the shell instructions used by the backend of the OpenMP compiler can be processed by invoking "/bin/sh -c" via. Runtime.exec(). However, Runtime.exec() does not support the I/O redirection of instructions. Therefore, MOMP intercepts redirection instructions first. Then, it uses InputStreamReader to retrieve the output of Runtime.exec() and writes or appends the retrieved output into the original destination files of the redirection instruction. For example, if the instruction is "ls >> file," MOMP will use Runtime.exec() to run "ls" first. Then, it will retrieve the output of Runtime.exec() and will append or write the output into file when the file is available or unavailable, respectively.

OpenMP allows users to set up the number of threads forked for the execution of parallel regions in their programs by using the OMP_NUM_THREADS environment variable. When the programs are executed, the runtime system of OpenMP retrieves the environment variable by calling the getenv() function to decide the number of forked threads. However, the environment variables in Android platforms are read only for user applications. To attack this problem, the OpenMP compiler of MOMP uses Java's ProcessBuilder to create a child process for program execution and sets a group of environment variables for the child process because ProcessBuilder can set up the environment variables for forked processes. When the child process executes user programs, the runtime system of MOMP is able to retrieve the environment variables set by the OpenMP compiler with calling the getenv() function.

During program compilation, the immediate files generated by the OpenMP compiler are stored in the/temp directory while this directory is not always available or accessible for any applications in any Android platforms. In Android, each application is allowed to access only its own directory such as /data/data/<packagename>/files. Consequently, the OpenMP compiler cannot access the root directory and the /tmp directory. For resolving this problem, MOMP creates a /tmp directory under /data/data/<MOMP>/files for the OpenMP compiler to store the immediate files of program compilation.

On the other hand, the OpenMP compiler of OMPICUDA supports OpenMP-to-CUDA but OpenMP-to-OpenCL translation. However, most of modern mobile devices support OpenCL but CUDA. Therefore, we implemented an OpenMP-to-OpenCL (simply denoted by OMPCL) translator for MOMP. Because most of CUDA API can be mapped onto OpenCL API, the OMPCL translator of MOMP mainly is implemented based on the framework of the OpenMP-to-CUDA translator of OMPICUDA. For considering paper length, we detailed only how the OMPCL translator maps the parallel regions of OpenMP to OpenCL kernels and the variables used in parallel regions from host memory to device memory for the execution of kernels as follows.

4.1.1. Mapping Parallel Regions to Kernels. OpenMP programming is based on a fork-join model while both of CUDA and OpenCL are based on a client-server model. It is necessary for the OMPCL translator to map the fork-join model onto the client-server one. The OMPCL translator implements the mapping of the two programming models as shown in Figure 4.

An OpenMP program basically consists of sequential regions and parallel regions. When an OpenMP program is translated into an OpenCL one, the sequential regions and parallel regions of the OpenMP program are mapped onto the host (i.e., client) program and kernels (i.e., server) of

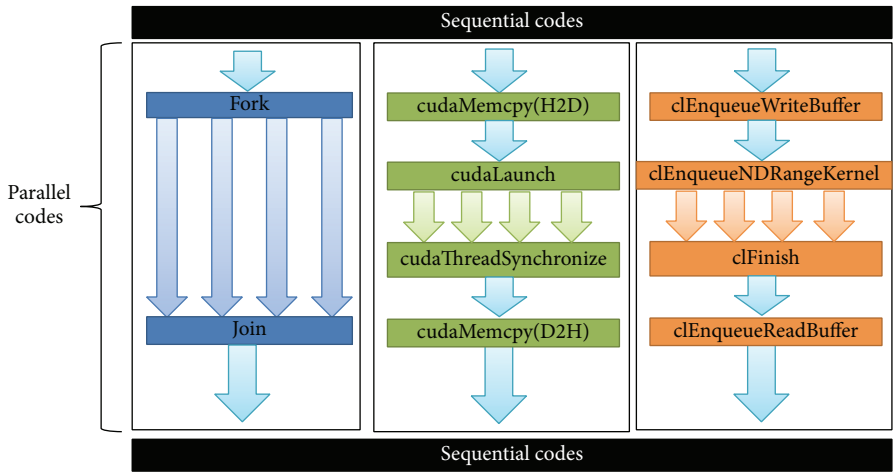

Figure 4: Programming model mapping between OpenMP and CUDA/OpenCL.

the OpenCL one, respectively. Whenever the execution of the host program arrives at a parallel region, it copies the shared data necessary for the parallel region from host to device by calling clEnqueueWriteBuffer() because device memory is independent to host memory. Then, it launches a kernel by calling clEnqueueNDRangeKernel() to device for concurrently processing the shared data in the device memory by multiple GPU threads. Finally, it copies the execution result of the kernel from device to host by invoking clEnqueueReadBuffer(). As previously described, the fork and join operations of OpenMP are mapped onto the clEnqueueNDRangeKernel() and clFinish() of OpenCL, respectively.

4.1.2. Mapping Variables from Host Memory to Device Memory. The working threads in a parallel region may access global, shared, and local variables. Global and shared variables are shareable for all the threads while local variables are not shareable, and they are duplicated for each thread. Accordingly, the OMPCL translator allocates shared and global variables on the global memory of GPU for data sharing among different thread blocks while it allocates local variables to the registers of GPU for each thread. To achieve this memory allocation policy, the OMPCL translator traverses all the global and shared variables that are accessed in the parallel region and marks the variables as GlobalShared and LocalShared. Because OpenCL does not support global variables and not all global variables are accessed in parallel region, the OMPCL translator creates two data structures, that is, GlobalShared and LocalShared to pass global and local shared variables into OpenCL kernels for GPU threads to access these shared variables. We used an example as shown in Figure 5 to explain how the OMPCL translator use these two data structures for mapping global and local shared variables from host memory onto device memory.

In this example, there are two one-dimensional matrices, that is, a and b, with 128 integer numbers. Because the two matrices are globally shared, the OMPCL translator defines a GlobalSahred data structure called globals to represent matrices a and b. In contrast, the c matrix declared in the main function is locally shared. Consequently, the OMPCL translator defines a LocalShared data structure called Locals to represent the matrix c. In addition, the OMPCL translator extracts the parallel region from the main function in Figure 5(a) to generate a kernel function called openclKernel with two parameters named as globals and locals, which are globals pointer and locals pointer, respectively. In addition, it changes the expression, that is, c[i]=a[i]+b[i] in the parallel region to be locals->c[i]=globals->a[i]+globals->b[i] as shown in Figure 5(b). On the other hand, the OMPCL translator generates a function called parallel0 for the host program to offload the parallel region onto device for execution as shown in Figure 5(c). In this function, the first step is to allocate one globals and locals data structure in device memory and then copy matrices a, b, and c from host memory to the globals and locals data structure in device memory. The next step is to create a kernel binding with the openclKernel function and then launch the kernel to device for execution. The final step is to copy the locals data structure from device memory to the c matrix in host memory.

On the other hand, the values of pointer variables used in parallel regions are invalid to device memory because of different address spaces. This problem is unable to be addressed at the time of program compilation because the values of pointer variables usually are determined and changed at runtime. To resolve this problem, the OMPCL translator generates a mapping table for tracking the information of host memory segments directed by pointer variables as shown in Figure 6. This mapping table is divided into global and dynamic sections because pointer variables may direct to global variables or dynamically allocated memory spaces. When an OpenMP program is compiled, the OMPCL translator searches global variables in the program and creates a registry_global_variable function for each global variable to store the start addresses and lengths of the global variable into

```
#include <omp.h>
#include <stdio.h>

int a[128],b[128];
int main() {
    int i;
    int c[128];
    #pragma omp parallel for     \
    localdim(128) globaldim(128) \
    device(conf, "test.conf")
    for(i=0; i<128; ++i)
        c[i]=a[i]+b[i];
    return 0;
}
```

(a) The original user program

```
// global variables
typedef struct GlobalShared{ int a[128],b[128]; }Globals;
// local variables
typedef struct LocalShared{ int c[128]; }Locals;
__kernel void openclKernel(
   Globals __global *globals
   Locals __global  *locals,
){
    // inductor and boundary of for-loop
    // lb: lower bound
    // ub: upper bound
    int i,lb=0,ub=128,step=1;
    // automatically cyclic-scheduling
    lb=step*get_global_id(0);
    step*=get_global_size(0);
    for(i=lb;i<ub;i+=step){
        // accesses to shared variables are rewritten
        locals->c[i]=globals->a[i]+globals->b[i];
    }
}
```

(b) The kernel program translated from (a)

```
#define clNewRWBuf(SIZE) \
   clCreateBuffer(context,CL_MEM_READ_WRITE,(SIZE),0,0)
#define clWriteBuf(D,POS,S,SZ) \
   clEnqueueWriteBuffer(commandQueue,(D),CL_TRUE,(POS),(SZ),(S),0,0,0)
#define clReadBuf(D,POS,S,SZ) \
   clEnqueueReadBuffer(commandQueue,(D),CL_TRUE,(POS),(SZ),(S),0,0,0)
#define addKerArg(arg)   \
   do{clSetKernelArg((kernel),argidx,sizeof((arg)),&(arg));++argidx;}while(0)
// global variables
typedef struct GlobalShared{ int a[128],b[128]; }Globals;
// local variables
typedef struct LocalShared{ int c[128]; }Locals;
void parallel0(void **argv){
  int *_pp_c=(int *)argv[0];
  size_t globaldim[3],localdim[3];
  cl_mem globals,locals;
  locals=clNewRWBuf(sizeof(Locals));
  globals=clNewRWBuf(sizeof(Globals));
  // copy data from host memory to the relative position in device memory
  clWriteBuf(locals,__offsetof(Locals,c),_pp_c,sizeof(int[128]));
  clWriteBuf(globals,__offsetof(Globals,a),a,sizeof(int[128]));
  clWriteBuf(globals,__offsetof(Globals,b),b,sizeof(int[128]));
  allocateCLThread(&globaldim,&localdim);
  // launch kernel
  { argidx=0;
    cl_kernel kernel=(clCreateKernel(ompclProgram,"openclKernel",0));
    // push arguments
    addKerArg(globals); addKerArg(locals);
    clEnqueueNDRangeKernel(commandQueue,kernel,1,0,globaldim,localdim,0,0,0);
    clFinish(commandQueue); clReleaseKernel(kernel);
  }
  // copy updated data from device memory to host memory
  clReadBuf(locals,__offsetof(Locals,c),_pp_c,sizeof(int[128]));
  // release allocated buffer
  clReleaseMemObject(locals);
  clReleaseMemObject(globals);
}
```

(c) The host program to (b)

Figure 5: Mapping variables from host memory to device memory.

the global section of the mapping table at run time. However, this method is not workable for dynamically allocated memory spaces. To resolve this problem, we added a hooker into dynamic allocation functions including malloc, valloc, calloc, and realloc in this paper. Whenever a dynamic allocation function is invoked, the hooker will fill the start address and length of allocation memory space into the dynamic section of the mapping table.

In addition to the mapping table, the OMPCL creates an exchange_pointer function for each pointer variable used in the parallel region before launching the kernel corresponding to the parallel region to device. The execution flow of the exchange_pointer function is as shown in Figure 7. The first is to search the mapping table to find out the value of the pointer variable (assume that it is addr) is located in which host memory segment. The second is to check if the devPtr field of the host memory segment is null, that is, whether the memory segment is mapped onto a device memory segment or not. If it is true, it is necessary to allocate a device memory space as big as the host memory segment and update the mapping table by filling the start address (assume that it is devaddr3) of allocated device memory space to the devPtr field copy first. Then, the next step is to copy data from the host memory segment to the allocated device memory space. The final is to change the value of the pointer variable to be devaddr3+(addr-a3). It is worth noting that the value of the pointer variable is changed back to addr by referencing the mapping table after the execution of the kernel finishes. By the above process, user programs can transparently access the same pointer variables in both of parallel regions and kernels.

4.2. A Reduced OpenCL Runtime Library Based on CUDA. MOMP translates OpenMP directives into OpenCL codes to exploit GPUs in mobile devices for data computation. However, some mobile devices such as MiPAD embedded with NVIDIA GPU support only CUDA driver but no OpenCL runtime library. As a consequence, the OpenCL programs generated by the MOMP compiler are unable to be

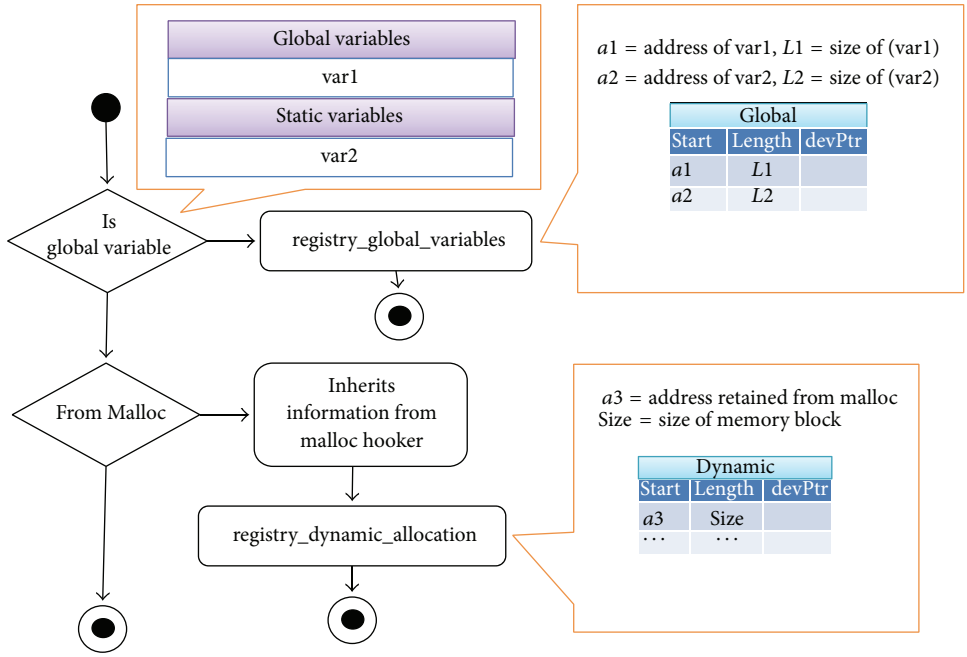

Figure 6: Registration of pointer variables.

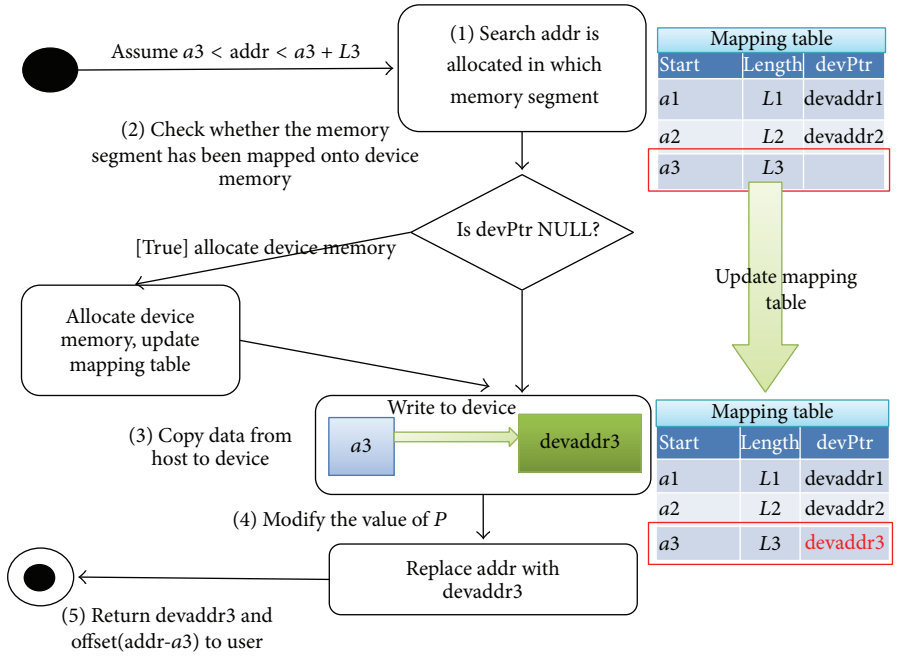

Figure 7: Operations of the exchange_pointer function.

executed by these mobile devices. To resolve this problem, we developed a reduced OpenCL runtime library called MCOCL (mobile CUDA-based OpenCL) based on CUDA driver for MOMP in this paper. Because MOMP provides users with OpenMP but OpenCL to develop parallel-computing applications in mobile devices, this reduced runtime library currently consists of only the OpenCL functions necessary for the OMPCL compiler to generate OpenCL source programs as shown in Table 1. We briefly describe the implementation of some of these functions as follows.

clGetDeviceIDs() is used to get the handles and count of devices available in a platform. In fact, the meaning of most of the parameters used in the CUDA driver API is as same as that used in the OpenCL API. For example, both of CUdevice and cl_device_id are used to obtain the information of devices. Consequently, MCOCL uses cuDeviceGetCount()

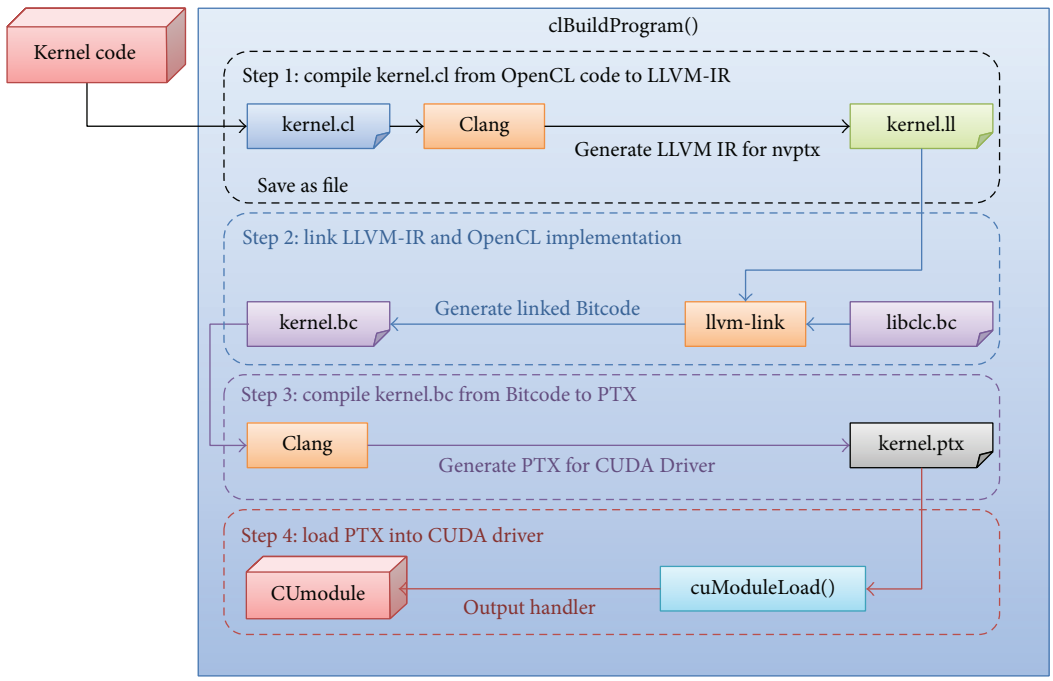

Figure 8: Execution flow of clBuildProgram() in MOMP.

Table 1: The OpenCL functions necessary for MOMP.

clGetPlatformIDs	clGetPlatformInfo
clGetDeviceIDs	clCreateContext
clGetContextInfo	clCreateProgramWithSource
clBuildProgram	clGetProgramBuildInfo
clCreateKernel	clEnqueueNDRangeKernel
clSetKernelArg	clCreateCommandQueue
clCreateBuffer	clEnqueueReadBuffer
clFinish	clEnqueueWriteBuffer
clReleaseKernel	clReleaseMemObject

and cuDeviceGet() to query the amount and identifiers of devices when clGetDeviceIDs() is called by user programs.

clCreateContext() is used to get a context in a given platform. An OpenCL context is created for one or more devices. It is used by the OpenCL runtime system for managing command queues, memory, programs, and kernels. By contrast, a CUDA context is created by cuCTxCreate() while it is useful for only one device. Consequently, MCOCL calls cuCtxCreate() to create a CUDA context for each device and store multiple CUDA contexts into a context array to represent one OpenCL context. When user programs intend to use different devices, MCOCL calls cuCtxPopCurrent() and cuCtxPushCurrent() to achieve context switch.

clCreateCommandQueue() is aimed at creating a command queue for launching kernels. Since OpenCL supports user programs to asynchronously execute the kernels appended in the command queue, MCOCL uses asynchronous CUDA streams to implement OpenCL command queues. When user programs call clEnqueueReadBuffer() or clEnqueueWriteBuffer() by a blocking mode, MCOCL invokes cuStreamSynchronize() to wait for the termination of a given kernel and then calls cuMemcpyHtoD() or cuMemcpyDtoH() to perform memory copy from host to device or from device to host. On the contrary, it invokes cuMemcpyDtoHAsync() or cuMemcpyHtoDAsync() when clEnqueueReadBuffer() or clEnqueueWriteBuffer() is called by a nonblocking mode. When user programs calls clFinish() to wait for the termination of kernels in the command queue, MCOCL calls cuStreamSynchronize() to wait for the termination of corresponding streams.

clBuildProgram() is to load a program source and then build a program executable from the program source. Although the cuModuleLoad() of CUDA is useful for loading programs, the loaded programs are composed of PTX (Parallel Thread Execution) codes instead of OpenCL sources. Therefore, MOMP must be able to translate OpenCL sources into PTX codes. Fortunately, LLVM provides libclc.bc to make Clang able to translate OpenCL into PTX for NVIDIA GPU. The execution flow of clBuildProgram() in MCOCL is as shown in Figure 8. The first step is to write OpenCL kernel codes into a file named as kernel.cl and compile this file by Clang to create another file called kernel.ll composed of LLVM IR. Because this kernel.ll file includes OpenCL symbols necessary to be supported by liblic, MCOCL uses llvm-link to link the kernel.ll with liblic to generate another file called kernel.bc and then uses Clang to translate the kernel.bc file into a PTX file called kernel.ptx. Finally, it calls cuModuleLoad() to load the PTX file into memory to be a CUmodule for clBuildProgram(). This CUmodule is regarded as program handle and is stored in cl_program. The messages

and parameters of program compilation are also stored into cl_program for calling clGetProgramBuildInfo() to retrieve these information later.

clSetKernelArg() is used to set the argument value for a specific argument of a kernel. This function is as same as the cuParamSetv() of CUDA. However, clSetKernelArg() uses *arg_index* to reference the augments of an OpenCL kernel while cuParamSetv() accesses the parameters of a CUDA kernel function by means of *offset*. In order to correctly convert *arg_index* into *offset*, MCOCL loads the kernel.ll as an IR module for LLVM and then invokes the getFunction() and getFunctionType() to retrieve the kernel from IR module and the prototype of the kernel, respectively. Then, it calls getParamType() and getTypeID() to extract the arguments from the kernel prototype and get the types of the arguments. Finally, it estimates the offset for ith argument by summating argument lengths from the first argument to the $(i-1)$th one and stores the offsets of all the arguments into the cl_kernel created by clCreateKernel() for translating *arg_index* to *offset* later.

clEnqueueNDRangeKernel() is used to submit a command for executing a given kernel. The purpose of this function is the same as that of cuLaunchKernel() while the thread configuration used for OpenCL kernels is different from that used for CUDA kernel functions. OpenCL uses global_work_size and local_work_size to represent the number of global work-items in work_dim dimensions and the number of work-items making up a work group for executing the kernel function, respectively. By contrast, CUDA uses gridDim and blockDim to represent the dimensions of a thread-block grid and a thread block, respectively. The local_work_dim of OpenCL can be regarded as the blockDim of CUDA. Consequently, MCOCL uses the local_work_dim to specify the dimensions of a thread block. On the contrary, it sets the dimensions of a thread-block grid by

$$\begin{aligned} \text{grid\_width} &= \left\lceil \frac{global\_work\_size\,[0]}{local\_work\_size\,[0]} \right\rceil, \\ \text{grid\_height} &= \left\lceil \frac{global\_work\_size\,[1]}{local\_work\_size\,[1]} \right\rceil. \end{aligned} \tag{1}$$

clCreateBuffer() is used to create a buffer object for data communication between host and device. To implement this function, MCOCL basically calls cuMemAlloc() and cuMemcpyHtoD() to allocate device memory for the buffer object and then copy data from host memory to the buffer object by default. However, user programs can set the *flags* argument of this function to specify the created buffer object is located at host or device memory. Consequently, MCOCL invokes different CUDA functions for clCreateBuffer() according to the *flags* augment. For example, it calls cuMemAllocHost() and memcpy() to allocate host memory for the buffer object and then copy the data to the allocated host memory when the *flags* argument is set as CL_MEM_ALLOC_HOST_PTR | CL_MEM_COPY_HOST_PTR.

Table 2: Experimental environment.

	XiaoMi MiPAD
OS	Android 4.4.4
Processor-CPU	Quad-Core ARM Cortex-A15 @ 2.2 GHz
Memory	2 GB LPDDR3
Cache	L1 cache 64 KB (32 KB I-cache, 32 KB D-cache) per core L2 cache Up to 4 MB per cluster
Processor-GPU	192 NVIDIA CUDA® Cores (NVIDIA Kepler™ architecture)
	SONY Z3
OS	Android 4.4.4
Processor-CPU	Qualcomm Snapdragon 801 ARMv7 @ 2.5 GHz
Memory	3 GB
Cache	L0: 4 KB + 4 KB, L1: 16 KB + 16 KB, L2: 2 MB
Processor-GPU	Adreno 330 (578 MHz)
	Samsung Note3
OS	Android 4.4.2
Processor-CPU	Quad-Core Snapdragon 800 Krait @ 2.3 GHz
Memory	3 GB LPDDR3
Cache	L0: 4 KB + 4 KB, L1: 16 KB + 16 KB, L2: 2 MB
Processor-GPU	Adreno 330 (578 MHz)
	InFocus M810
OS	Android 4.4.4
Processor-CPU	Qualcomm Snapdragon 801 ARMv7 @ 2.5 GHz
Memory	2 GB
Cache	L0: 4 KB + 4 KB, L1: 16 KB + 16 KB, L2: 2 MB
Processor-GPU	Adreno 330 (578 MHz)
	Test AP
Matrix multiplication	Matrix size $(N * N)$, N = {512, 1024, 1536, 2048}
Nbody	Body number = {1024, 2048, 3072, 4096}, Loop = 100
SOR	Matrix size $(N * N)$, N = {1024, 2048, 3072, 4096}, Loop = 100
Mandelbrot set	Image size $(N \times N)$, N = {256, 512, 768, 1024}

5. Performance Evaluation

We have evaluated the performance of MOMP in this paper. Our experimental environment includes Xiaomi MiPAD, Sony Z3, Samsung Note3, and InFocus M810. The resource configurations of these four mobile devices are listed in Table 2. We developed four applications including matrix multiplication (MM), Successive over Relaxation (SOR), Nbody, and Mandelbrot set. Both of MM and Nbody have lots of data computation while MM requires more memory space than Nbody. By contrast, the computation demand of SOR is much less than MM and Nbody while the memory demand of SOR is more than that of Nbody but less than that of MM. Mandelbrot set is an application, which generates and draws fractal images based on recursive formulas. It requires less memory space than MM and SOR but more

than Nbody. Since Nbody and SOR are iterative applications, the runtime system of MOMP forks multiple threads and assigns the threads to multicore CPU or launches kernels to GPU for concurrent execution at the beginning of each iteration. When the execution processor is GPU, it copies input data from host to device before a kernel is launched and copies output data from device to host after the kernel is terminated for each iteration. Basically, this performance evaluation was to individually run the test applications on the four mobile devices and estimate the execution time of the test applications by using CPU or GPU in the mobile devices.

On the other hand, we estimated the performance of memory accesses and memory copy between host and device (denoted as HtoD and DtoH) in mobile devices before we did our experiments. The experimental result is shown in Figure 9. The performance of memory accesses in MiPAD is worse than that of the others. For memory copy between host and device, MiPAD is the worst of the four mobile devices. By contrast, Z3 is the best, and M810 almost is as good as Z3. As to Note3, it is better than MiPAD but worse than Z3 and M810. We will use this result to explain the execution performance of the four mobile devices later.

5.1. Performance of MOMP Using CPU in Mobile Devices. Our first experiment was aimed at evaluating the performance of MOMP when it used CPU in mobile devices for executing the test applications. We ran the test applications by forking 1, 2, and 4 threads to estimate the execution time of the test applications executed by 1, 2, and 4 CPU cores (denoted as OMP-1core, OMP-2core, and OMP-4core), respectively. To measure the overhead of runtime system in MOMP, we created a sequential version for the test applications by deleting the OpenMP directives in the original source programs and executed the sequential test applications by mobile devices. The experimental result is shown in Figures 10, 11, 12, and 13. Basically, the execution performance of the OpenMP programs is better than that of the sequential-C ones when the core number is larger than one. The performance of all the test applications is significantly improved with the increase of core number no matter which mobile device is used for program execution. However, the execution performance of Mandelbrot is not improved well when the core number is increased from 2 to 4 because the computation of this application is not evenly distributed over four working threads.

On the other hand, the runtime system overhead of MOMP is negligible for the performance of the MM, Nbody, and Mandelbrot applications. However, it is significant for the performance of the SOR application especially when it is executed on MiPAD or Note3. Our performance profile shows that when the SOR application is executed with multiple threads, the numbers of cache misses, instructions, and branches are significantly increased. It is worth noting that the performance of the MOMP-1core case is better than that of the sequential-C case when the MM and Mandelbrot applications are executed with MiPad. There are two reasons for this result. The first is that the speed of memory access of MiPAD is slower than the others as shown in Figure 9. In other words, the penalty cost of cache misses in MiPAD is obvious especially. The second is that the OpenMP compiler replaces the parallel region in the main function by a function call. This is helpful for minimizing the size of the main function and the miss rate of instruction cache. Consequently, the OpenMP compiler can save the overhead of cache misses for the MM and Mandelbrot applications in MiPAD. However, this situation does not appear in the other mobile devices because the penalty cost of cache misses becomes much smaller. It also disappears when the other applications are executed in MiPAD because the granularity of the main function in the Nbody and SOR application is smaller.

In this experiment, we also compared MOMP with CCTools. Since CCTools uses gcc to compile user programs, the executable files generated by CCTools are native codes. By contrast, MOMP uses Clang to compile user programs into LLVM IRs and converts IRs by MCJIT into native codes for processors to execute the programs at runtime. In this performance comparison, we ran the MM application with the problem size of 2048×2048 float-point numbers, the Nbody application for 4096 particles, the SOR application for the problem size of 4096×4096 float-point numbers, and the Mandelbrot application for 1024×1024 pixels, respectively. The result of performance comparison between MOMP and CCTools is as depicted in Figure 14. The execution performance of MOMP is better than that of CCTools for all the test applications in MiPAD because MOMP can optimize the executable codes of user programs according to the architecture of ARM Cortex-A15 in MiPAD while CCTools cannot. Moreover, MOMP is more efficient for the Nbody and Mandelbrot applications than CCTools in most of the mobile devices. Conversely, CCTools performs more efficiently than MOMP for the SOR application in Z3, M810, and Note3 because the computational demand of the SOR application is small, and MOMP has to spend extra time on converting LLVM IRs into native codes at runtime.

5.2. Performance of MOMP Using GPU in Mobile Devices. Our second experiment was aimed at evaluating the performance of MOMP when it used the GPU of mobile devices for executing the test applications. We ran the same test applications with GPU and estimated the execution time of the applications. The breakdown of the execution time of the test programs is shown in Figures 15, 16, 17, and 18. In these figures, the BuildPro label represents the cost of clBuildProgram() which was called for generating a kernel function for each parallel region in the test applications. In addition, the HtoD and DtoH labels denote the cost of host-to-device and device-to-host memory copy, respectively. The execution label denotes the execution time of the kernel launched by the test programs.

For the MM application, the performance of MOMP in MiPAD is better than that in the others. Although MOMP spent more time on program building in MiPAD because the cost of memory accesses in MiPAD is high, it spent much less time on data computation in MiPAD because the computation speed of MiPAD is fast. The same situation also happens in the Nbody application. Because Nbody is a computation intensive but data-intensive problem, almost all the execution time of this application on the mobile devices except MiPAD

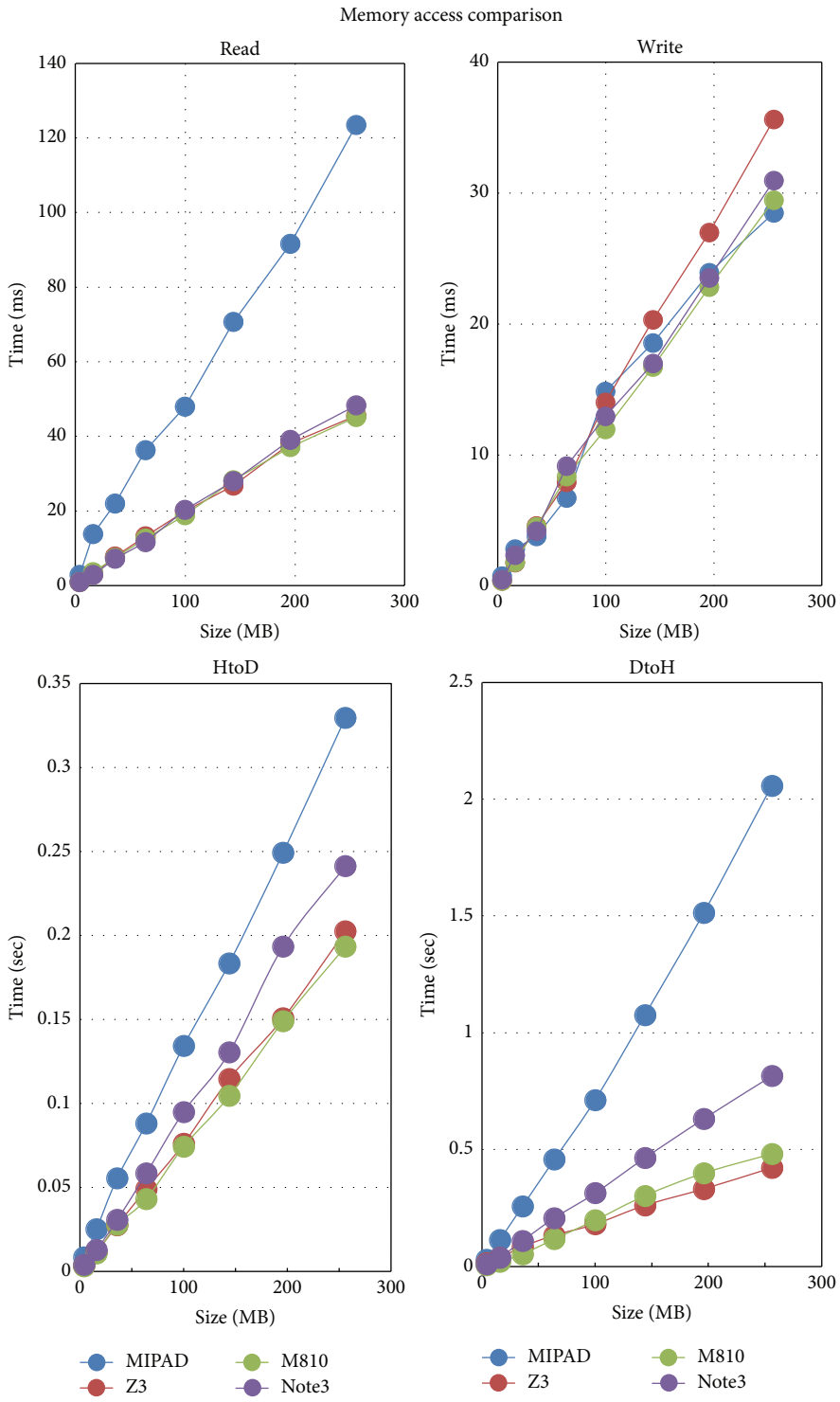

Figure 9: Performance of memory accesses in mobile devices.

is spent on the execution of the kernel. As to Mandelbrot, its execution performance is mainly dependent on the cost of memory access and copy when it is executed by MiPAD. By contrast, its execution time is determined by the cost of data computation when it is executed by the other mobile devices. Different from the previous applications, SOR requires a large amount of data transfer between host and device while the amount of data computation is relatively much less. Consequently, MiPAD spends more time on memory copy between host and device but less time on the execution of the SOR kernel than the other devices. Because the increased cost of memory copy is more than the saved cost of kernel execution,

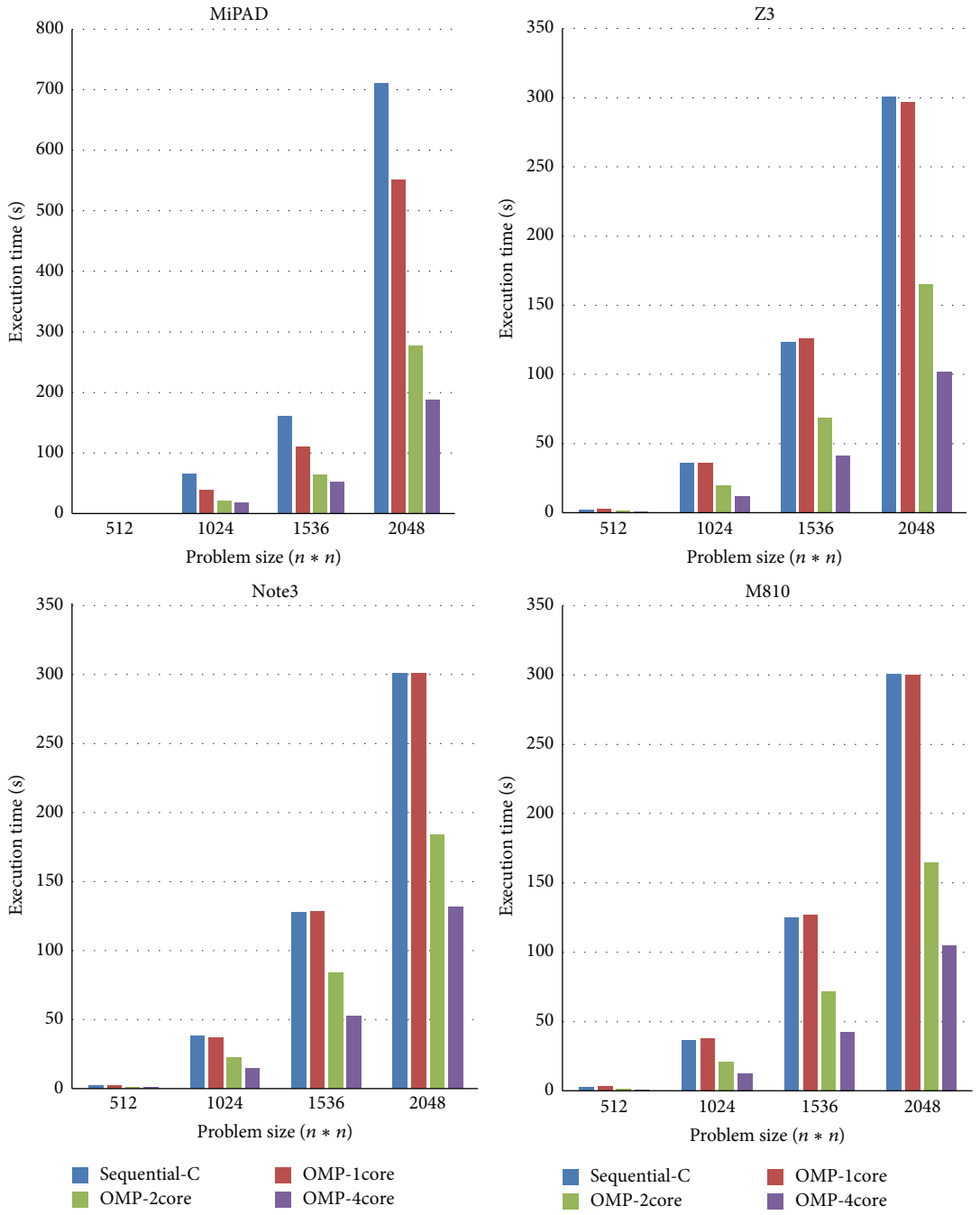

Figure 10: Performance of MM executed by CPU in mobile devices.

the execution performance of MiPAD is worse than those of the other mobile devices for the SOR application.

On the other hand, we divided the CPU execution time by the GPU execution time of the test applications to estimate the performance improvement obtained by replacing CPU with GPU, as shown in Figure 19. For the MM application, the GPUs of Z3, Note3, and M810, respectively, provide 5, 5, and 4 times speedup in the best case. For the Nbody application, the maximal performance difference between GPU and CPU is 10, 9, and 7 times when MOMP uses Z3, Note3, and M810, respectively. The SOR application obtains 2.5, 2, and 1.8 speedups, respectively, from the GPUs of M810, Z3, and Note3. However, the performance comparison also shows that when the problem size is small, the GPU of MiPAD does not contribute performance improvement for the test applications compared with the CPU of MiPAD. The main reason is that MiPAD has to spend a long period of time on program building and data transfer between host and device for GPU while it does not need to do these things for CPU. The benefit from saving computation time

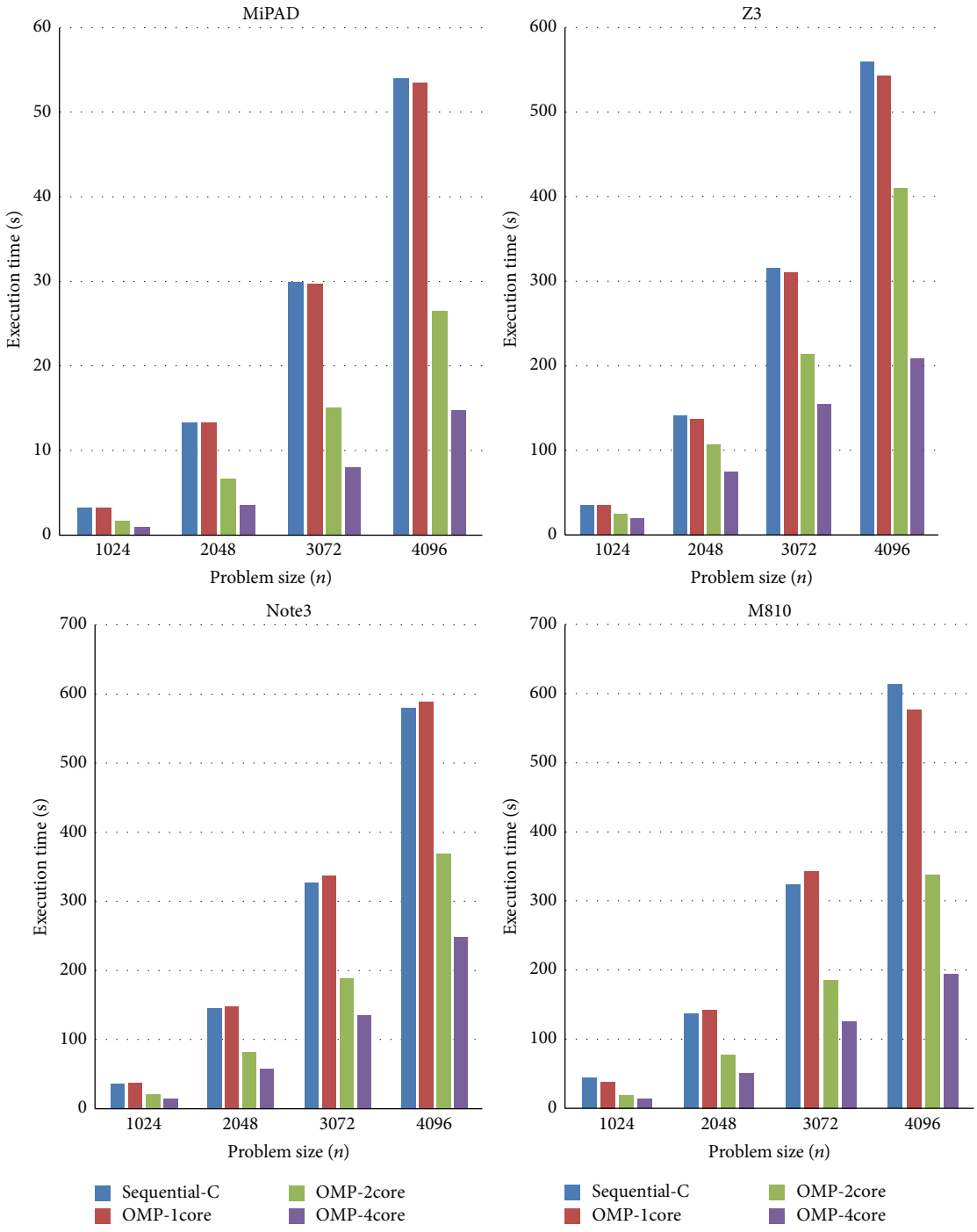

Figure 11: Performance of Nbody executed by CPU in mobile devices.

is seriously reduced by the cost of program building and data transfer. Consequently, MiPAD does not provide performance improvement as well as the other mobile devices for the MM and Nbody applications and degrades the execution performance of the SOR application when it replaces CPU with GPU for program execution. Conversely, Mandelbrot obtains a great performance improvement by replacing CPU with GPU no matter which mobile device is used because it has high parallelism and massive data computation but a few of data communication between host and device.

5.3. Impact of Resource Selection. Our third experiment was dedicated to evaluating the impact of resource selection on the performance of user applications. We created an application with two parallel regions that are aimed at resolving the MM and SOR problems, respectively. We ran the two parallel

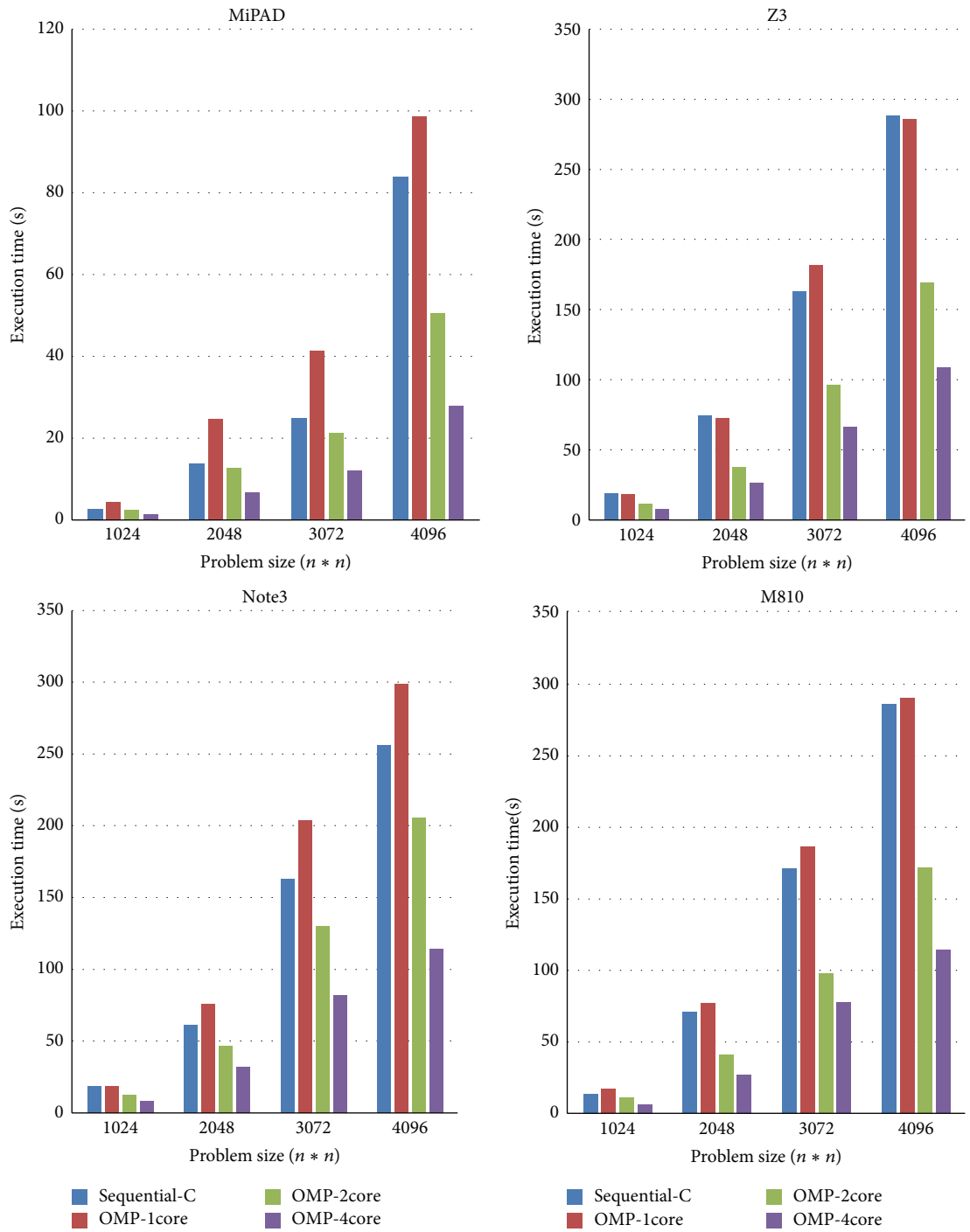

Figure 12: Performance of SOR executed by CPU in mobile devices.

regions of the application in MiPAD by three ways of resource selection. The first way is to select CPU executing the two parallel regions while the second way is to select GPU. The third way is to select GPU for the parallel region of the MM problem but CPU for the parallel region of the SOR problem. The experimental result is shown in Figure 20.

We can find that the third way of resource selection is the best for the performance of this application that has different properties in different parallel regions. By contrast, the first way is the worst because the MM application is computation intensive while it is executed by CPU. Conversely, the second way is better than the first but worse than the third because it selects GPU for the parallel region of SOR while the SOR problem is suitable to be executed by CPU but GPU. The previous discussion implies that selecting proper processors for executing different parallel regions in the same application is

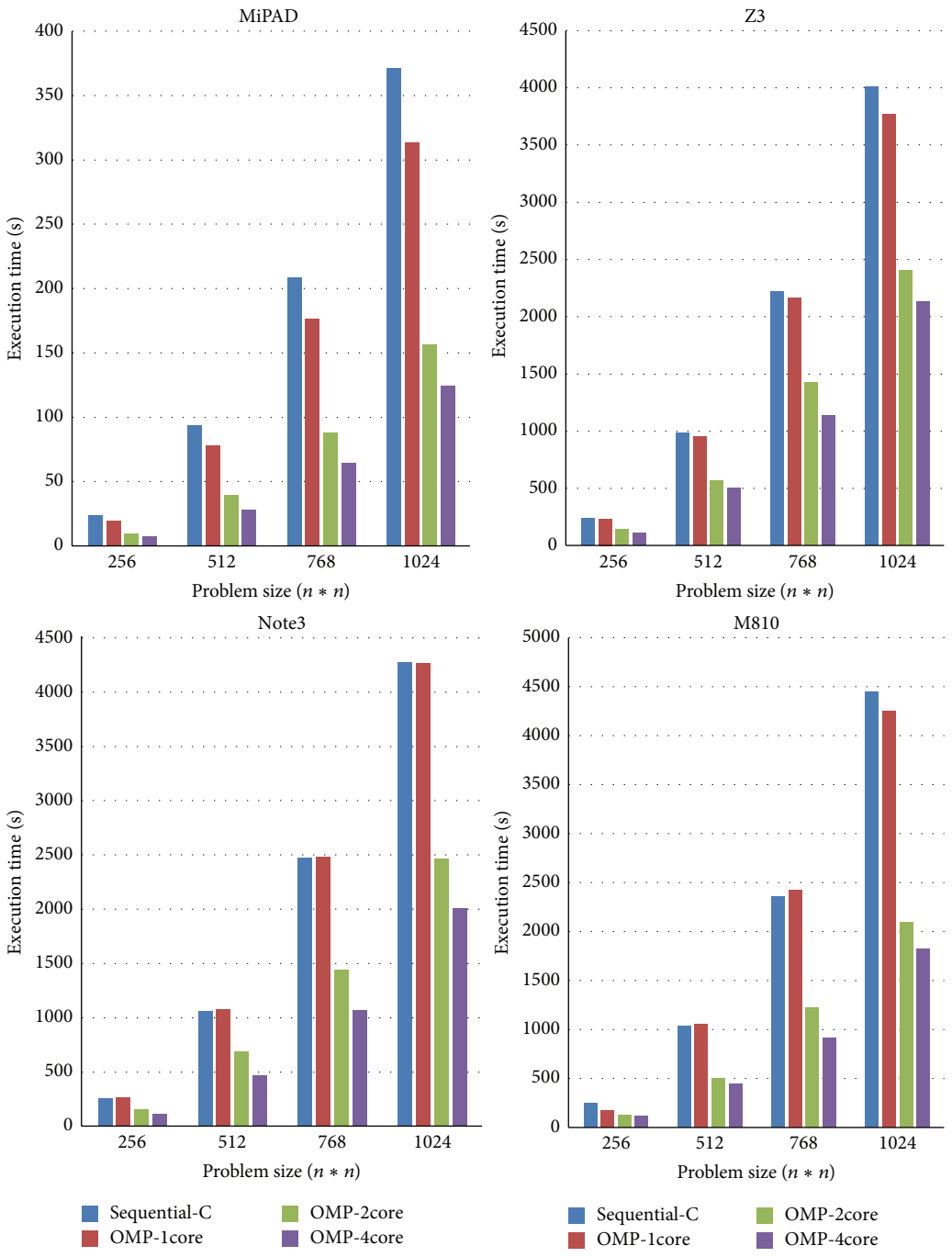

Figure 13: Performance of Mandelbrot executed by CPU in mobile devices.

very important and necessary for the execution performance of the application. MOMP provides an easy interface, namely, target(pthread or opencl), for users to address this issue.

6. Related Work

In recent years, some researches have been done for reducing the complexity of parallel programming on mobile devices. We discussed these researchers as follows.

Android-Aparapi [26] supports users to exploit mobile CPUs or GPUs for parallel processing data in Java. It can automatically translate parallel Java bytecodes into OpenCL hosts and OpenCL kernels or the codes of using Java Thread Pool (JTP). It executes user programs with GPU by default. However, if OpenCL driver or GPU is not supported on mobile devices, it will use multicore CPUs to execute user programs. It is developed based on Aparapi [27] while it optimizes the execution performance of Aparapi on Android. For example,

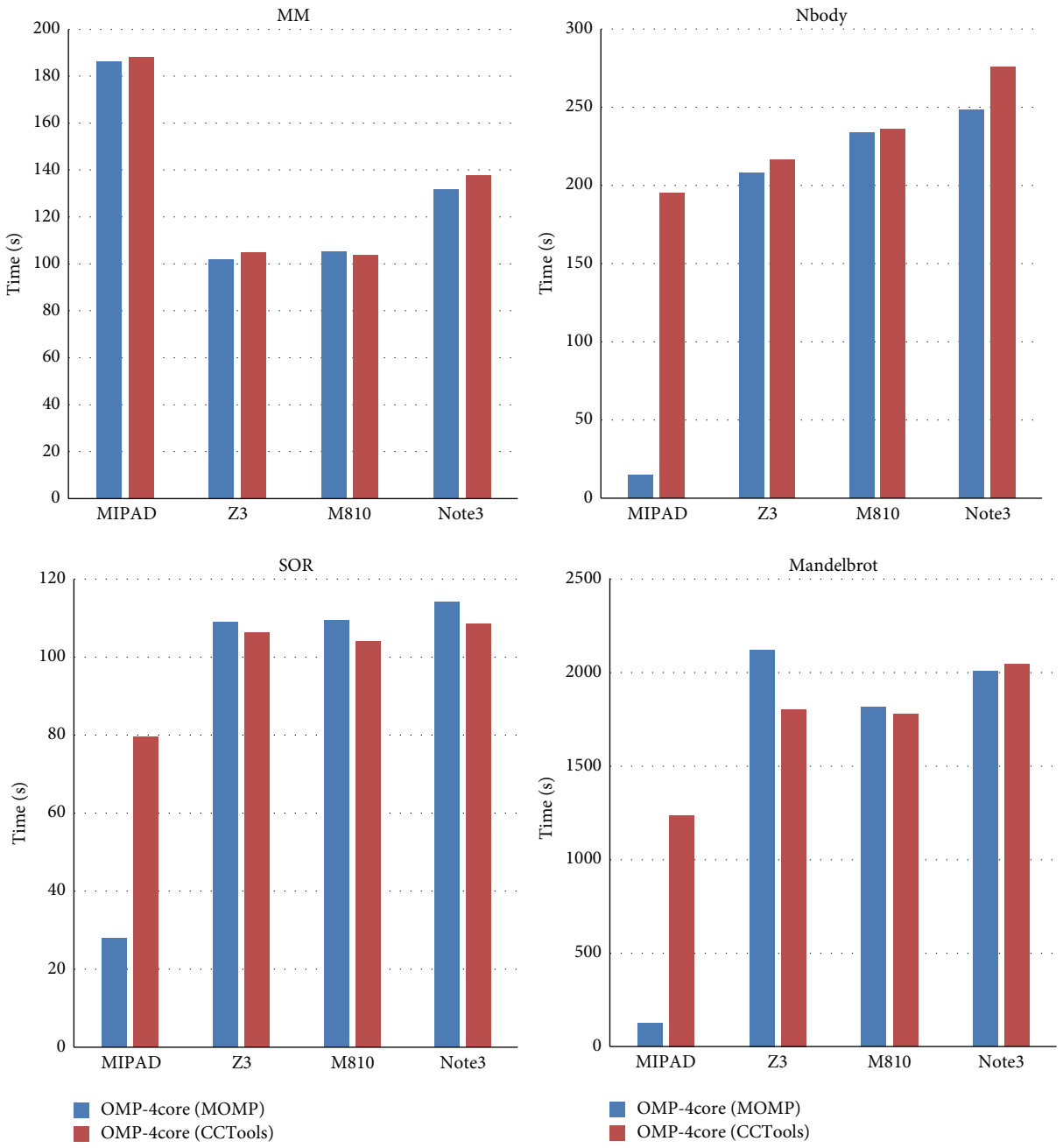

Figure 14: Performance comparison between MOMP and CCTools.

it exploits Byte Buffer for data communication with OpenCL kernels while minimizing the overhead of garbage collection. It also tries to gather many computational instructions into a single JNI call to minimize the overhead of Dalvik VM (DVM) and OpenCL runtime. Although Android-Aparapi successfully enables Java to be a uniform API for mobile CPU and GPU, it is necessary to modify DVM for Android-Aparapi. This implies that user applications are not portable to any Android platform unless the modified DVM is installed on target mobile devices. In addition, the front end of Android-Aparpi must be executed at PC or workstation for passing the classes of user programs to the backend at mobile devices for generating OpenCL kernels and executing the kernels with mobile GPU. As a result, Android-Aparapi cannot allow users to directly develop applications on mobile devices anytime and anywhere without network connection.

Pyjama [27] is a Java compiler and runtime system for supporting OpenMP-like directives on Android. They make use of JavaCC [28] to implement a source-to-source translator for converting OpenMP to Java. Since the fork-join model of OpenMP results in that GUI thread delays to pick up user events, it proposes a GUI-aware directive called *freeguithread* to automatically handle the synchronization between GUI thread and the threads of parallel regions.

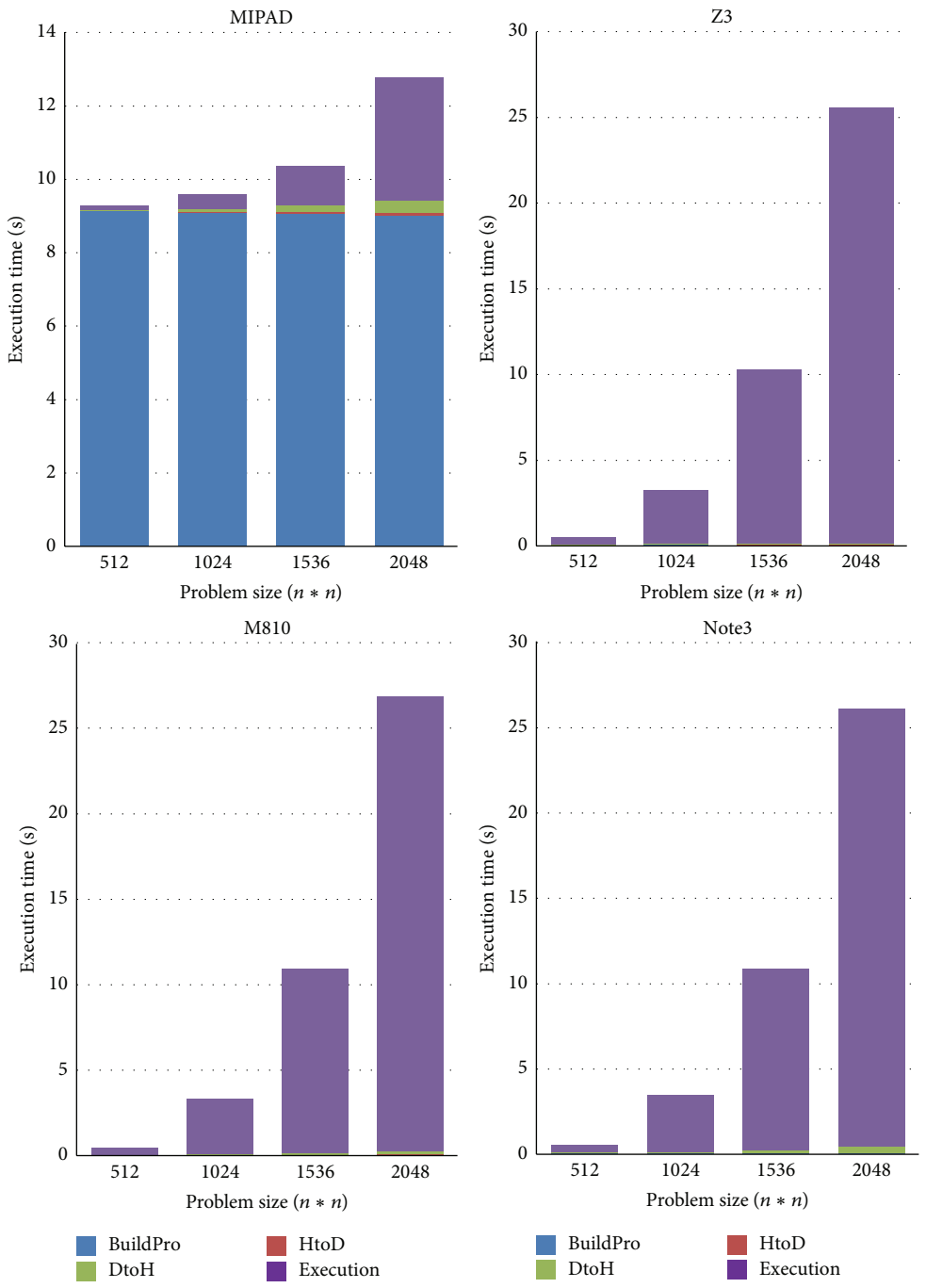

Figure 15: Performance of MM executed by GPU in mobile devices.

Although Pyjama provides an easy API, that is, OpenMP-like directives and GUI-aware directives for users to increase the parallelism of their mobile applications and maintain the responsiveness of the applications, it cannot exploit the computational power of GPUs in mobile devices to increase the execution performance of mobile applications.

CCTools is an integrated development environment on mobile devices. The main advantage of this APP is allowing users to directly edit, compile, and execute OpenMP programs on mobile devices through an embedded Linux terminal. It makes use of gcc to compile users' OpenMP programs. However, the executable files generated by gcc are runnable for CPU but GPU in mobile devices. Since it requires users to manually type commands to edit, compile, and execute their programs in a text-mode terminal, it is not friendly and convenient for those who are not used to operating Linux OS.

HIPA (Heterogeneous Image Processing Acceleration) [29] is a domain specific language for speeding up image processing on embedded systems and mobile devices. HIPA enables users to design the kernels of image processing by C-like language while it automatically converts the kernels into CUDA/OpenCL/Renderscript/Filterscript ones to process

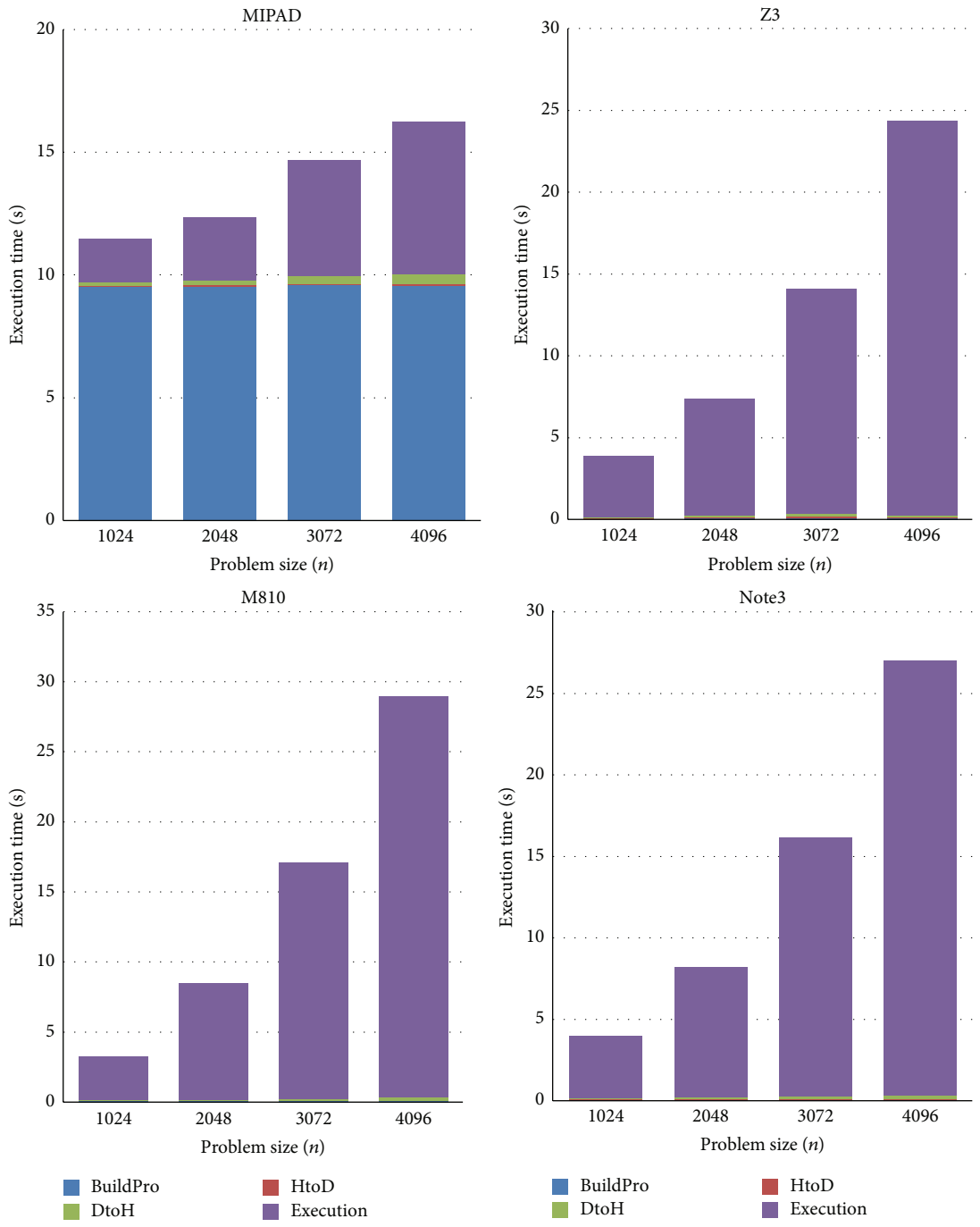

Figure 16: Performance of Nbody executed by GPU in mobile devices.

images with GPU. In order to exploit GPU efficiently, it makes use of the memory hierarchy of GPU to minimize the cost of data accesses. For instance, it stores the data shared by all the threads in the same kernel at global memory and texture memory while allocating the data shared by the threads of the same thread block at shared memory or local memory. When kernels are translated by means of Renderscript/Filterscript, the expressions of accessing images are translated to call rsGetElementAt function. Moreover, HIPA optimizes the OpenCL implementation for HSA (Heterogeneous System Architecture). Since host memory and device memory shares the same region on HSA, HIPA uses mmap() and munmap() to minimize the cost of data transfer between host and device. With the support of HIPA, users need not deal with memory allocation and data transfer and select execution processor through environment variables. However, HIPA is dedicated for image processing but general-purpose computing.

Although the above toolkits can effectively reduce the parallel programming of multicore CPU or manycore-GPU on mobile devices, they require users to compile programs

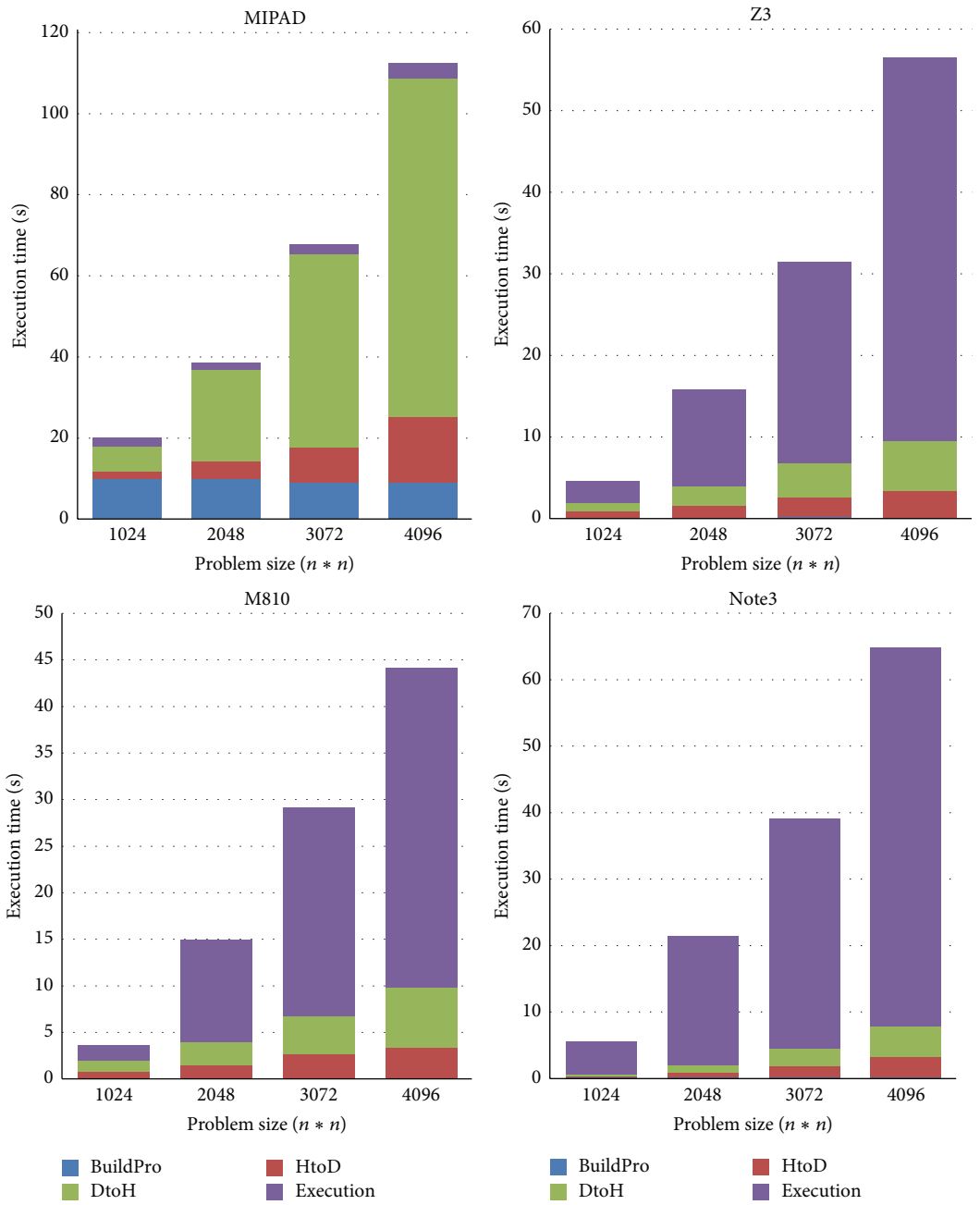

Figure 17: Performance of SOR executed by GPU in mobile devices.

on computers and then move the executable files from computers to mobile devices for execution except CCTools. Moreover, most of them are not able to support a uniform programming interface for users to exploiting CPUs and GPUs in mobile devices. By contrast, MOMP provides a complete and friendly programming environment for users to develop parallel programs in mobile devices anytime anywhere without network connection. Furthermore, it supports a uniform programming interface, that is, OpenMP, to reduce the parallel programming complexity of multicore CPU and many-core GPU on mobile devices at the same time.

7. Conclusions and Future Work

In this paper, we have successfully developed a mobile OpenMP programming environment called MOMP. Using this APP, users can develop OpenMP applications to exploit the computational power of CPU and GPU in mobile devices for resolving their problems anytime and anywhere without the assistance of remote servers. In addition, they can easily move their OpenMP programs from computers to mobile devices without modifying their programs. It is convenient for them to switch their computing environment between

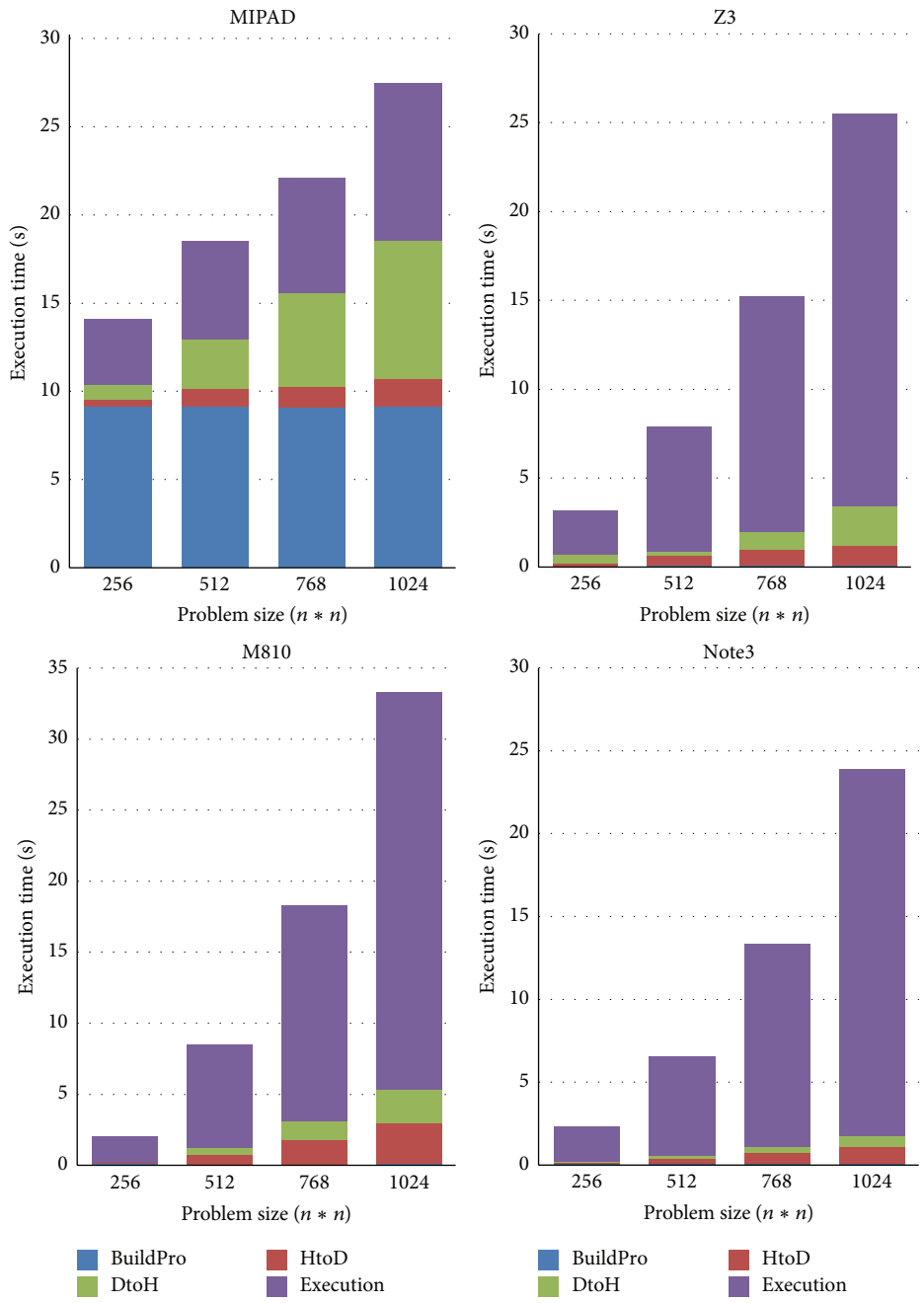

Figure 18: Performance of Mandelbrot executed by GPU in mobile devices.

computers and mobile devices. On the other hand, our experimental results have shown that MOMP can effectively exploit the computational power of mobile devices for the execution performance of user programs, and it performs more efficiently than CCTools for the test applications in most of the experimental mobile devices. It is worth noting that carefully selecting CPU and GPU is essential for the execution performance of user programs.

Although mobile devices recently have become more powerful in data computation, the high complexity of parallel programming always was a big problem for users to move their computation platform from computers toward mobile devices. In this work, MOMP has successfully reduced the programming complexity of heterogeneous computing in mobile devices because OpenMP is much easier than CUDA, OpenCL, and Pthread. This contribution is useful to encourage users to move their working environment toward mobile devices because they can save lots of time and effort on learning new programming interface and application development. Although MOMP currently supports only OpenMP 2.0, it is enough for users to develop lots of data-parallelism applications with parallel loops. In past decades, OpenMP programming has been popularly and widely applied in many research areas such as high energy physics, bioinformatics,

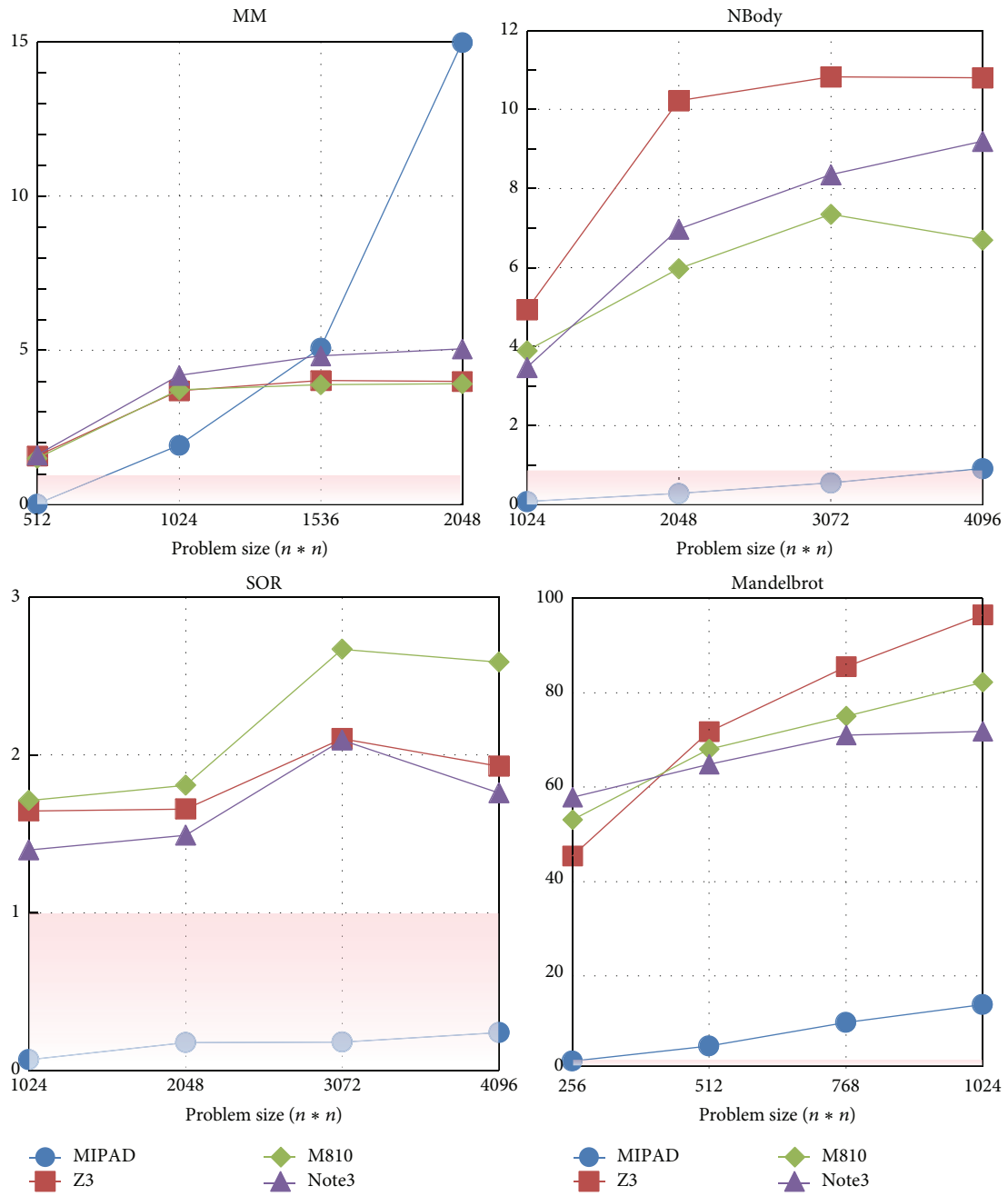

Figure 19: Speedup gained by replacing CPU with GPU in mobile devices.

data mining, earthquake prediction, and multimedia processing and is also an important course for the department of computer science and engineering in universities. The appearance of MOMP is helpful for researchers and students to make use of mobile devices for increasing their work efficiency and learning effect because mobile devices are more affordable and easier to carry for users than Notebooks.

Although the computational power of mobile devices has been greatly improved, mobile devices still cannot work as long as PCs because of finite battery capability. If user programs cannot finish their work before they use up the battery power of mobile devices, the context of these programs should be logged into storages in order to resume their execution later, or they must be executed from beginning. Accordingly, we will develop a backup/recovery mechanism and a live migration scheme to increase the reliability of MOMP. In addition, we will extend MOMP to support OpenMP 3.0 and 4.0 when OpenCL 2.0 is available in mobile devices.

Competing Interests

There is no conflict of interests regarding the publication of this paper.

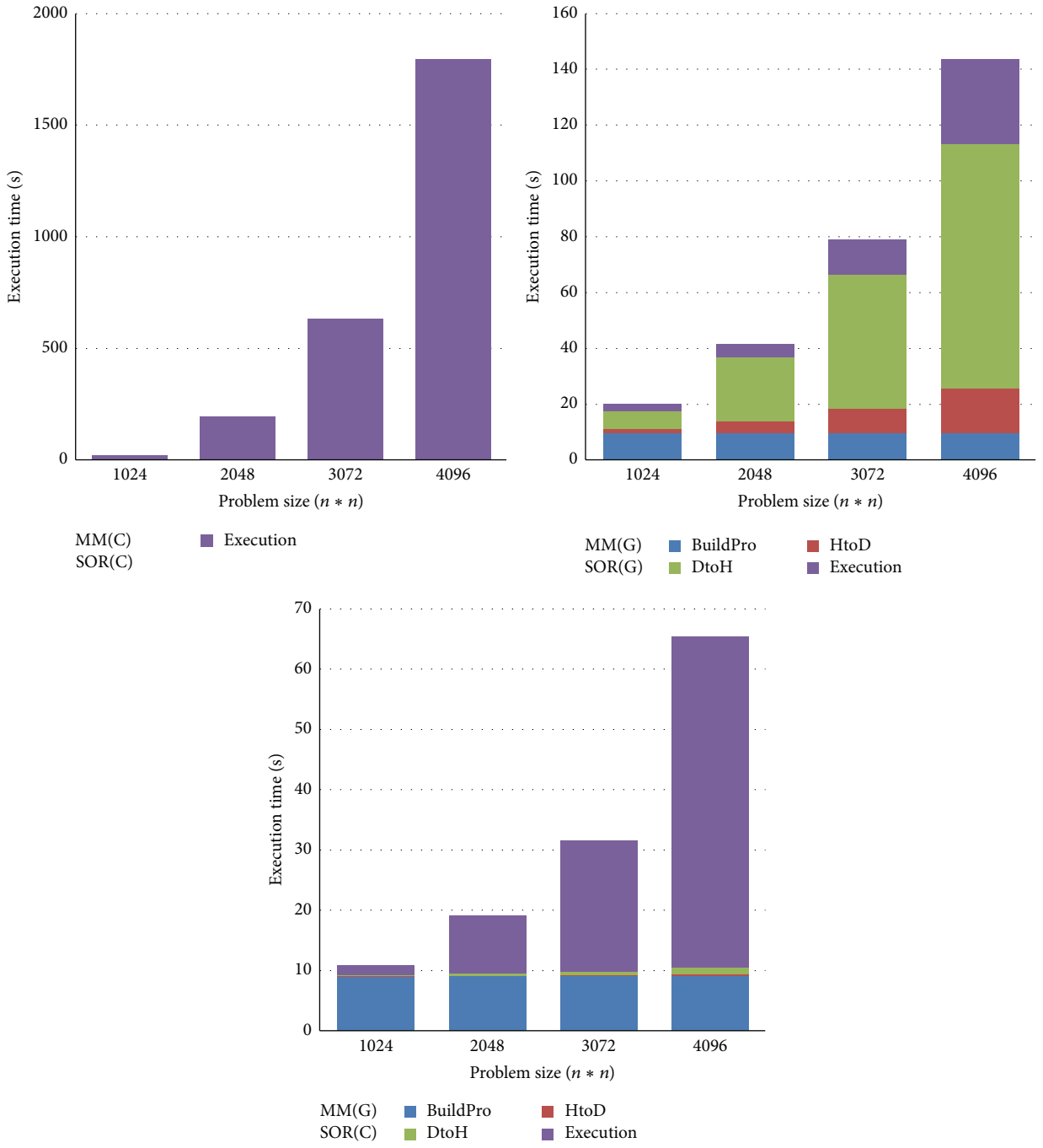

Figure 20: Impact of resource selection in mobile devices.

Acknowledgments

This work is supported by Ministry of Science and Technology of Republic of China under the research Project no. MOST 103-2221-E-151-044.

References

[1] E. W. John, GUN Octave, 2015, http://www.gnu.org/software/octave/index.html.

[2] Corbin Champion.: Addi–Matlab/Octave clone for Android, 2015, https://code.google.com/p/addi.

[3] MathWorks.: MATLAB—The language of technical computing, 2015, http://www.mathworks.com/products/matlab.

[4] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.

[5] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science & Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[6] NVIDIA, *CUDA C Programming Guide*, 2015, http://docs.nvidia.com/cuda/cuda-c-programming-guide.

[7] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: a parallel programming standard for heterogeneous computing systems," *Computing in Science & Engineering*, vol. 12, no. 3, Article ID 5457293, pp. 66–73, 2010.

[8] n0n3m4.: C4droid - C/C++ compiler & IDE, 2015, https://play.google.com/store/apps/details?id=com.n0n3m4.droidc.

[9] Arduino team, CppDroid – C/C++ IDE, 2015, https://play.google.com/store/apps/details?id=name.antonsmirnov.android.cppdroid.

[10] A. Chukov, "*CCTools*," 2015, https://play.google.com/store/apps/details?id=com.pdaxrom.cctools.

[11] J. Hoeflinger, "(Intel).: *Cluster OpenMP for Intel*[5] *Compilers*," 2010, https://software.intel.com/en-us/articles/cluster-openmp-for-intel-compilers.

[12] T.-Y. Liang, S.-H. Wang, C. E.-K. Shieh, C.-M. Huang, and L.-I. Chang, "Design and implementation of the OpenMP programming interface on linux-based SMP clusters," *Journal of Information Science and Engineering*, vol. 22, no. 4, pp. 785–798, 2006.

[13] C. Amza, A. L. Cox, S. Dwarkadas et al., "TreadMarks: shared memory computing on networks of workstations," *Computer*, vol. 29, no. 2, pp. 18–28, 1996.

[14] J.-B. Chang, C.-K. Shieh, and T.-Y. Liang, "A transparent distributed shared memory for clustered symmetric multiprocessors," *The Journal of Supercomputing*, vol. 37, no. 2, pp. 145–160, 2006.

[15] H. Bae, D. Mustafa, J.-W. Lee et al., "The Cetus source-to-source compiler infrastructure: overview and evaluation," *International Journal of Parallel Programming*, vol. 41, no. 6, pp. 753–767, 2013.

[16] S. Lee and R. Eigenmann, "OpenMPC: extended OpenMP for efficient programming and tuning on GPUs," *International Journal of Computational Science and Engineering*, vol. 8, no. 1, pp. 4–20, 2013.

[17] G. Noaje, C. Jaillet, and M. Krajecki, "Source-to-source code translator: OpenMP C to CUDA," in *Proceedings of the IEEE 13th International Conference on High Performance Computing and Communications (HPCC '11)*, pp. 512–519, Banff, Canada, September 2011.

[18] *HMCSOT: OpenMP to OpenCL Source-to-Source Translator*, 2015, http://www.openfoundry.org/of/projects/1117.

[19] T. Y. Liang and Y. J. Lin, "JCL: an OpenCL programming toolkit for heterogeneous computing," in *Grid and Pervasive Computing: 8th International Conference, GPC 2013 and Colocated Workshops, Seoul, Korea, May 9–11, 2013. Proceedings*, vol. 7861 of *Lecture Notes in Computer Science*, pp. 59–72, Springer, Berlin, Germany, 2013.

[20] A. Barak and A. Shiloh, "*The VirtualCL (VCL) Cluster Platform*," 2015, http://www.mosix.org/vcl/VCL_wp.pdf.

[21] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters," in *Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12)*, pp. 341–351, ACM, Istanbul, Turkey, June 2012.

[22] T. Y. Liang, H. F. Li, and Y. C. Chen, "A ubiquitous integrated development environment for C programming on mobile devices," in *Proceedings of the 12th IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC '14)*, pp. 184–189, Dalian, China, August 2014.

[23] Clang Team, "*Clang: a C language family Frontend for LLVM*," 2015, http://Clang.llvm.org/.

[24] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO '04)*, pp. 75–86, March 2004.

[25] H.-F. Li, T.-Y. Liang, and J.-Y. Chiu, "A compound OpenMP/MPI program development toolkit for hybrid CPU/GPU clusters," *The Journal of Supercomputing*, vol. 66, no. 1, pp. 381–405, 2013.

[26] H.-S. Chen, J.-Y. Chiou, C.-Y. Yang et al., "Design and implementation of high-level compute on Android systems," in *Proceedings of the 11th IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia '13)*, pp. 96–104, IEEE, Montreal, Canada, October 2013.

[27] K. G. Gupta, N. Agrawal, and S. K. Maity, "Performance analysis between aparapi (a parallel API) and JAVA by implementing sobel edge detection Algorithm," in *Proceedings of the National Conference on Parallel Computing Technologies (PARCOMPTECH '13)*, pp. 1–5, Bangalore, India, February 2013.

[28] *JavaCC is a Parser/Scanner Generator*, 2015, https://java.net/projects/javacc.

[29] R. Membarth and O. Reiche, "*HIPAcc*," 2015, http://hipacc-lang.org/.