OpenMDSP: Extending OpenMP to Program Multi-Core DSP

Jiangzhou He*, Wenguang Chen*, Guangri Chen†, Weimin Zheng*, Zhizhong Tang* and Handong Ye†
*Tsinghua National Laboratory for Information Science and Technology (TNList)
Tsinghua University, Beijing, China
Email: hejz07@mails.tsinghua.edu.cn, {cwg, zwm-dcs, tzz-dcs}@tsinghua.edu.cn
†Huawei Technologies Co. Ltd., Shenzhen, China
Email: {chenguangri, hye}@huawei.com

***Abstract*—Multi-core Digital Signal Processors (DSP) are widely used in wireless telecommunication, core network transcoding, industrial control, and audio/video processing etc. Comparing with general purpose multi-processors, the multi-core DSPs normally have more complex memory hierarchy, such as on-chip core-local memory and non-cache-coherent shared memory. As a result, it is very challenging to write efficient multi-core DSP applications.**

The current approach to program multi-core DSPs is based on proprietary vendor SDKs, which only provides low-level, non-portable primitives. While it is acceptable to write coarse-grained task level parallel code with these SDKs, it is very tedious and error prone to write fine-grained data parallel code with them.

We believe it is desired to have a high-level and portable parallel programming model for multi-core DSPs. In this paper, we propose OpenMDSP, an extension of OpenMP designed for multi-core DSPs. The goal of OpenMDSP is to fill the gap between OpenMP memory model and the memory hierarchy of multi-core DSPs.

We propose three class of directives in OpenMDSP:(1) data placement directives allow programmers to control the placement of global variables conveniently; (2) distributed array directives divide whole array into sections and promote them into core-local memory to improve performance, and (3) stream access directives promote big array into core-local memory section by section during a parallel loop's processing.

We implement the compiler and runtime system for OpenMDSP on FreeScale MSC8156. Benchmarking result shows that seven out of nine benchmarks achieve a speedup of more than 5 with 6 threads.

***Keywords*-OpenMP; multi-core DSP; data parallelism; LTE;**

I. Introduction

A. Expressing data parallelism is important for future multi-core DSP systems

Multi-core Digital Signal Processors (DSP) are widely used in wireless telecommunication, core network transcoding, industrial control, audio and video processing, etc. Comparing with general purpose multi-processors, the multi-core DSPs usually have more complex memory hierarchy, such as on-chip core-local memory and non-cache-coherent shared memory [1]. The on-chip core-local memory are usually addressable with different address spaces and can not be accessed by other cores directly. The non-cache-coherent shared memory also makes memory management much more complex for shared data since programmers are required to maintain the coherence of data manually. As a result, it is very challenging to write efficient multi-core DSP applications.

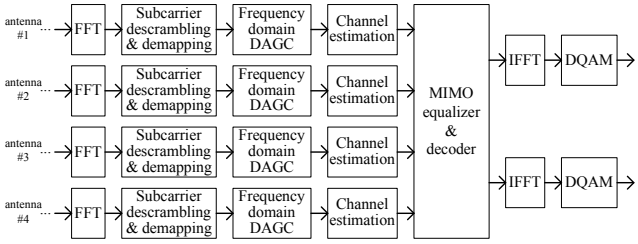

Figure 1. Partial task graph of a LTE base station physical layer uplink.

The state-of-the-art approach to program multi-core DSPs is based on proprietary vendor SDKs, which only provides low-level, non-portable primitives. It is a common practice to use these SDKs to provide coarse-grained task-level parallelism for applications. For example, the next generation of wireless telecommunication protocol, Long Term Evolution (LTE) [2], is a very important application of multi-core DSPs. Currently, developers parallelize LTE base station applications in task level. Fig. 1 is a partial task graph of LTE base station physical layer uplink. Input of each Fast Fourier Transform (FFT) task is the carrier waveform received from each antenna port. Waveforms of different carriers are fed into FFT tasks one after another. Developers map tasks for waveform from different antennas to different cores to exploit task parallelism, or map tasks of different stages to different cores to exploit pipeline parallelism between adjacent carriers, or both.

Although task parallelism and pipeline parallelism seem to work well, they are not sufficient. The emergence of low-power multi-core DSPs demands programmers to exploit data parallelism as well. For example, the PicoChip PC205 [1] has 248 cores and works at frequency of 280MHz. While the low frequency gives the chip a big power efficiency boost over traditional multi-core DSPs,

```
__attribute__((section(".m3_shared"))) int a[1024];
__attribute__((section(".m3_shared"))) int sum = 0;
__attribute__((section(".m3_shared"))) spinlock l1;
void barrier(int count) { ... }
void sum_a() {
  int thread_id = get_core_id() - 2;
  int num_threads = 4;
  int local_sum = 0;
  int lower = 1024 * thread_id / num_threads;
  int upper = 1024 * (thread_id + 1) / num_threads;
  if (thread_id == 0) sum = 0;
  barrier(num_threads);
  for (i = lower; i < upper; i++)
    local_sum += a[i];
  acquire_spinlock(&l1);
  sum += local_sum;
  release_spinlock(&l1);
  barrier(num_threads);
}
```

Figure 2. Data parallelized code for array sum, by SDK of FreeScale MSC8156. sum_a() is intended to run on core #2 – core #5. Contents of barrier() is omitted.

it also imposes more challenges to programmers. Without employing data parallelism in applications, it is very difficult for this chip to meet the latency requirement of LTE signal processing. Thus, we believe it is critical to support data parallelism as well as task parallelism for future multi-core DSP platforms.

B. Problems of Current Programming Models for Multi-core DSPs

Today's common practice to program multi-core DSPs in industry is to use the low-level, proprietary vendor SDKs. While they provide reasonable abstraction for task parallelism and pipeline parallelism, they are usually too low-level when used to express data parallelism. An example of data parallel code written with the SDK of FreeScale MSC8156 is shown in Fig. 2.

From the example, we can see that programmers need to write code for barrier, loop dividing and scheduling, data reduction, data sharing and synchronization manually, which makes programming very tedious and error prone. The other issue is portability. The language extension defined in these proprietary SDKs are not portable to DSPs of other vendors.

Researchers have proposed some programming models to solve these problems. SoC-C [3] is one programming model for heterogeneous multi-core systems. It has well support for task parallelism and pipeline parallelism, but lacks support for data parallelism. StreamIt [4] is another influential programming model for stream applications. It defines an elegant dataflow programming model to exploit data parallelism, task parallelism and pipeline parallelism. The problems with StreamIt are two folds: 1) It is not easy to incrementally change the existing code to StreamIt. Significant part of legacy code may need to be re-written. 2) StreamIt is based on the static data-flow model. While in DSP applications, there are scenarios which require dynamic level of parallelism and task binding.

C. Extend OpenMP to support multi-core DSP programming

The purpose of this research is to provide a parallel programming model for multi-core DSPs which features: 1) Allow incrementally change to support data parallelism on existing task/pipeline parallel code written with SDKs. 2) High level abstractions to avoid tedious loop bound calculation and synchronization for data parallel code. 3) Portable among different DSP platforms.

We propose to extend OpenMP to support multi-core DSP programming. OpenMP [5] is a widely adopted industrial standard. Extension based on OpenMP provides a good chance for portability. OpenMP is powerful to express data parallelism. It provides high level abstractions to prevent programmers from tedious work such as calculating parallel loop bounds for each thread. With OpenMP, programmers can add a few annotations on sequential code to parallelize it incrementally, which matches our goal of incrementally parallelization very well.

Standard OpenMP only supports cache coherent shared memory systems. In multi-core DSPs, there are core-local memory and non-cache coherent shared memory which imposes two key challenges to OpenMP on multi-core DSP:

Core-local memory: On general purpose multi-core CPU systems, shared auto variables residing in the stack of master thread can be easily accessed by other worker threads because of a unified address space, but on most multi-core DSPs, the core-local memory has different memory spaces with other memory hierarchy. Stack resides on core-local memory and cannot be accessed from other cores. If we need to put shared auto variables in the stack, we need a way to make it accessible to other threads.

Non-cache coherent shared memory: On general purpose multi-core CPU systems, the shared memory are cache coherent which could boost the speed of shared data accessing when the data are not really shared during a period of time, the accesses will be cache hit without accessing the main memory. However, the shared memory of multi-core DSPs is not cache coherent, so every access to the variables in shared memory should be a slow shared memory access if there's no hint to put some data into the fast core-local memory. We need to address this issue in our extension design.

In this paper, we present OpenMDSP, an extension of OpenMP 2.5 for multi-core DSP. We have also implemented an OpenMDSP compiler and runtime system for FreeScale MSC8156, a DSP processor with six cores. Our paper makes three main contributions:

- We make standard OpenMP 2.5 programs run correctly on multi-core DSPs without modification. We design and implement compiler transformation and runtime

systems so that all standard OpenMP 2.5 directives and APIs remain the same semantics on multi-core DSPs. This work allow us to annotate current sequential code of tasks to support data parallelism.

- To improve the performance of OpenMP code, we propose OpenMDSP, which is an extension of OpenMP 2.5, to allow optimized usage of the complex memory hierarchy of multi-core DSP systems. Especially, we support core-local memory and non-cache coherent shared memory with three new class of directives of OpenMDSP:
  (1) **Data placement directives** allow programmers to control the placement of global variables conveniently;
  (2) **Distributed array directives** divide whole array into sections and promote them into core-local memory to improve performance, and
  (3) **Stream access directives** promote big array into core-local memory section by section during a parallel loop's processing.
- We implement the compiler and runtime system for OpenMDSP on FreeScale MSC8156. Nine benchmarks are used to evaluate the performance of OpenMDSP. Seven out of nine benchmarks achieve a speedup of more than 5 with 6 threads.

The rest of the paper is organized as follows. Section II introduces the architecture of multi-core DSP in more detail. Section III defines the features of OpenMDSP. Section IV states the design and implementation of OMDPFS. We evaluates the implementation in Section V, and discusses the limitations and alternative solutions in Section VI. Section VII reviews related works, and Section VIII concludes the paper.

II. Overview of Multi-core DSP Architecture

Multi-core DSP is the target hardware platform for OpenMDSP. FreeScale MSC8156 is one typical high performance DSP with six cores. As shown in Fig. 3 [6], a chip of FreeScale MSC8156 consists of six SC3850 DSP cores. The memory hierarchy consists of four levels:

L1 ICache and L1 DCache: the first level instruction cache and data cache private to each core. They are transparent to software. Each has a size of 32KB.

L2 Cache and M2 Memory: the second level memory with a total size of 512KB private to each core. It can be configured to divide into L2 cache and M2 memory. L2 is transparent to software while M2 is addressable core-local memory.

M3 Memory: on-chip shared memory memory with a size of 1056KB. The latency is larger than L1, L2 and M2.

Main Memory: off-chip shared memory accessed by two DDR controller, always with the largest latency. Six cores, M3 memory, DDR controllers, DMA, I/O controller and other blocks are connected with the Chip-Level Arbitration and Switching System (CLASS).

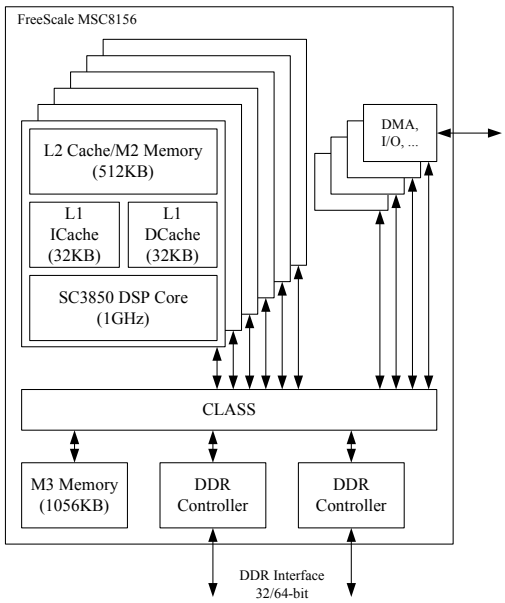

Figure 3. Architecture blocks inside a FreeScale MSC8156 chip.

Programmers can decide whether cache is enabled or disabled for particular linking sections. FreeScale MSC8156 does not provide hardware managed cache coherence among different cores, that means either cache should be disabled for shared sections, or cache coherence should be maintained by software.

Although OMDPFS is designed for FreeScale MSC8156, OpenMDSP is intended to work for general multi-core DSPs. The memory hierarchy of most multi-core DSP contains core-local memory, on-chip shared memory and off-chip shared memory [1], that is the fundamental assumption of OpenMDSP for underlying hardware platform.

III. OpenMDSP Language Extensions

OpenMDSP is designed based on OpenMP 2.5, and inherits the execution model, memory model, directives and API functions defined in OpenMP 2.5 specification. As stated in Section I, to expose the memory hierarchy for programmers, we have extended OpenMP by a few new directives as shown in Fig. 4.

A. Extension for Data Placement

With OpenMP directives, programmers cannot specify mapping between variables and memory levels. In OpenMDSP, by default, shared variables with static storage duration are placed in on-chip shared memory, and threadprivate variables are placed in core-local memory. However, sometimes programmers needs to change the default placement, for example, core-local memory may not have enough space to carry all threadprivate variables, hence some of

alloc-directive **::=**
  #pragma domp alloc (*place* **,** *var-list*) ***new-line***
place **::= corelocal** | **chipshare** | **offshare**
var-list **::=** ***variable*** | ***variable* ,** *var-list*
defaultalloc-directive **::=**
  #pragma domp defaultalloc *defalloc-placedef*$^+$ ***new-line***
defalloc-placedef **::= normal** (*place*) | **threadprivate** (*place*)
distribute-directive **::=**
  #pragma domp distribute (*var-list*) *dist-clause** ***new-line***
    statement
dist-clause **::= size** (***expression***) | **peek** (***expression* , *expression***)
  | **copyin** | **copyout** | **bulk** (***expression***) | **nowait**
for-respect-directive **::=**
  #pragma domp for respect (*variable*) *forres-clause** ***new-line***
    for-statement
forres-clause **::= private** (*var-list*) | **firstprivate** (*var-list*)
  | **lastprivate** (*var-list*) | **reduction** (***operator*** **:** *var-list*)
  | **nowait**
omp-parallel-for-directive [REDEFINE]**=**
  #pragma omp parallel for ***omp-parallel-for-clause**** ***new-line***
  *stream-directive**
    for-statement
omp-for-directive [REDEFINE]**=**
  #pragma omp for ***omp-for-clause**** ***new-line***
  *stream-directive**
    for-statement
stream-directive **::=**
  #pragma domp stream (*var-list*) *stream-clause** ***new-line***
stream-clause **::= size** (***expression***) | **rate** (***expression***)
  | **peek** (***expression* , *expression***) | **copyin** | **copyout**
  | **nowait**

Figure 4. OpenMDSP syntax extensions. *omp-parallel-for-directive* and *omp-for-directive* are redefinitions for *omp for* directive defined in OpenMP specification, while others are definitions for new directives.

the threadprivate variables need to reside in on-chip shared memory or off-chip shared memory.

We introduce *alloc* directive for data placement. *place* can be *corelocal*, *chipshare* or *offshare* to indicate core-local memory, on-chip shared memory or off-chip shared memory, respectively.

We introduce another directive *defaultalloc* to specify the default data placement location. *defaultalloc* directive takes effect for all variables with static storage duration defined after that till the next *defaultalloc* directive which overrides the setting.

B. Extension for Distributed Array

Shared data is stored in on-chip shared memory or off-chip shared memory, which has a considerable latency. It is hard to optimize access for shared array by OpenMP features. We have observed that in a class of parallel algorithms, such as matrix operations, each thread only need to access a portion of an array. We have defined *distribute* directive for such situation.

distribute directive is used to create distributed duplications for shared array in core-local memory during execution of its following statement. Any access to such array in the following statement is done by the private duplication, which improves the performance. The whole array is divided into n contiguous sections, one for each thread, where n is the number of threads. For each thread, we call the corresponding section as the *main section* for this thread. In the statement followed by *distribute* directive each thread is only allowed to access its main section of an array unless a *peek* clause is present.

Variables placed in *distribute* clause is the name of the shared array or a pointer to the first element of the array. If a pointer is placed here, *size* clause is required to specify the size of the array it pointed to. *size* clause is not needed for static array.

peek clause allows one thread to read some elements before and after the main section. The two expressions should specify number of elements allowed to read before and after. This clause is designed for algorithms in which access ranges for an array in adjacent iterations have intersection.

copyin clause makes data in shared data copied into private memory before execution of *statement*. Without *copyin*, the initial value of the private duplication is undefined. *copyout* clause makes data copied back to shared memory, and the original values in shared memory do not change when *copyout* is not specified. *copyout* only takes effect for the main section, while *copyin* also takes effect for the amount of elements as specified in *peek* clause.

bulk clause is used to specify the minimal grain for array range division. The size of each main section is guaranteed to be a multiple of the value of specified expression except the last section. The default minimal grain is 1.

Like *parallel*, *for* and *single* directives defined in OpenMP, *nowait* is used to remove the implicit barrier after execution of *statement*.

OpenMDSP provides another directive, *for respect*, which allows programmers to write code to iterate through the main section of a distributed array conveniently. Effect of *domp for respect* directive is much like *omp for* directive, which parallelize the following *for-statement*. Like *omp for* directive, *private*, *firstprivate*, *lastprivate*, *reduction* and *nowait* clauses are applicable. The distinct feature of *domp for* is that it divides loop range consistent with the distribution of the array specified inside the braces after "respect". More precisely, the loop range in each thread is guaranteed to be the intersection of the main section of this thread for the specified array and the whole loop range. Fig. 5 presents an example for *distribute* and *for respect* directives.

C. Extension for Stream Access

Although distributed array is helpful to reduce access latency for shared array, it cannot work if the main section is too large to reside in core-local memory. For access patterns applicable to *distribute* and *for respect*, shared arrays can also be fetched in core-local memory or written back part by part as needed. That is the purpose of *stream* directive.

```
double a[M][N];
double b[N][P];
double c[M][P];
void compute_c() {
  int i, j, k;
  double s;
#pragma omp parallel firstprivate(b)
#pragma domp distribute(a) copyin
#pragma domp distribute(c) copyout
#pragma domp for respect(a)
  for (i = 0; i < M; i++) {
    for (j = 0; j < P; j++) {
      s = 0;
      for (k = 0; k < N; k++)
        s += a[i][k] * b[k][j];
      c[i][j] = s;
    }
  }
}
```

Figure 5. Matrix multiplication, an example of *distribute* and *for respect*. "M", "N" and "P" are macros defined as integer constants. Note that in C language, a two dimensional array is an array of one dimensional array. so each element of array "a" and "c" is a row of the matrix. The two *distribute* directives divide the matrices by rows, and the *for respect* directive takes effect for the following loop on the row number of matrix "a". Besides, OpenMDSP guarantees distributions of two distributed arrays with the same (*array-size*/*bulk-size*) values to be identical, so it is also safe to access array "c" by "i".

```
void complex_mul(int n, float* r, const float* a,
                 const float* b) {
  int i;
#pragma omp parallel for
#pragma domp stream(a, b) rate(2) copyin
#pragma domp stream(r) rate(2) copyout
  for (i = 0; i < n; i++) {
    r[i*2] = a[i*2]*b[i*2] - a[i*2+1]*b[i*2+1];
    r[i*2+1] = a[i*2]*b[i*2+1] + a[i*2+1]*b[i*2];
  }
}
```

Figure 6. Complex vector multiplication, an example of *stream* directive. "M" is a macro defined as integer constant.

We extend *omp parallel for* and *omp for* directives defined in OpenMP specification by *stream* directives as their suffix. The semantics of these directives without suffix *stream* directive do not change.

Syntax of *stream* directive is much like *distribute* directive. Variables specified after *stream* should be arrays or pointers to array elements. *stream* is used to indicate that these arrays are accessed in a stream pattern. Programmers can use *rate* clause to specify number of elements accessed in an iteration.

Like *distribute* directive, *stream* directive also maps shared data to core-local memory. Instead of dividing the whole array, *stream* directive maps the shared array by *windows*, one window for one chunk of the parallel loop. If *copyin* is present, data in the next window will be copied into core-local memory before each chunk, and if *copyout* is present, data in the previous window will be written back after each chunk. *stream* directives change the default chunk size of parallel loop. If the chunk size is not specified in *schedule* clause, the default chunk size is defined by implementation. Implementation of OpenMDSP should guarantee core-local memory is able to carry each window. Fig. 6 gives an example of *stream* clause.

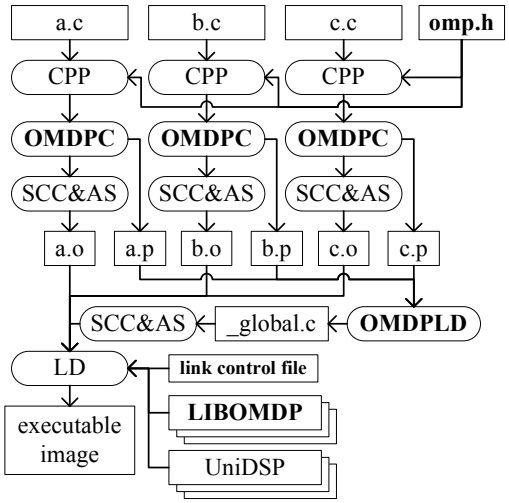

Figure 7. Compilation process of OpenMDSP application in OMDPFS.

Although *distribute* and *stream* act in a similar way, their design goals differ. *stream* is good at boosting performance for arrays too big to distribute into core-local memory. If an array is accessed in several parallel loops, *distribute* is a better choice.

IV. Design and Implementation

A. Overview of Compilation Process

Fig. 7 shows the compilation process of an OpenMDSP application in OMDPFS. For a OpenMDSP source file named `a.c`, the driver of our OpenMDSP compiler first feeds it into C preprocessor (CPP). The output of CPP is the input of OMDPC, the source-to-source OpenMDSP compiler of OMDPFS. OMDPC transforms OpenMDSP directives into bare C code, which is fed into Starcore C Compiler (SCC) and the assembler (AS). SCC and AS are tools in FreeScale MSC8156 SDK. The output of AS is the relocatable object file `a.o`.

Besides, OMDPC generates another file, `a.p`, which contains summary information of `a.c`. OMDPLD synthesizes these summary files and generates `_global.c`, the global data file. We will discuss content of the global data file in Section IV-C.

The driver calls the linker to link all object files together with LIBOMDP and UniDSP library, finally generate the executable image. LIBOMDP is the runtime library of OMDPFS. It is based on UniDSP, a DSP operating system

developed and internally used by Huawei Technologies Co. Ltd. At runtime, each core load the same executable image for execution, this is due to the SPMD nature of OpenMP.

OMDPC and LIBOMDP are primary parts of OMDPFS, while the header file `omp.h`, OMDPLD and the link control file are also provided by OMDPFS.

B. Overall Design

1) Fulfill the Execution Model: In UniDSP, *task* is the basic scheduling unit. A UniDSP task can only run on the core which creates it, and never migrate to other cores. In OMDPFS, we map OpenMDSP threads to tasks on different cores.

For each core which loads the OpenMDSP image, we create one task to perform the job of one OpenMDSP thread. Entry of this task is a function in LIBOMDP. Because all cores execute the same image, LIBOMDP decides what to do by ID of current core. Task running on core with the least ID is treated as the master thread. In this task, the transformed entry function of application is executed at startup. Tasks running on other cores pend on a semaphore until the master thread encounters a parallel region and post the semaphore to notify them.

2) Fulfill the Memory Model: As stated before, OpenMDSP inherits the memory model of OpenMP, which is a relaxed-consistency, shared memory model [5]. In multi-core CPU and SMP, since cache coherence protocols guarantees sequential consistency for memory operations, which is stricter than relaxed-consistency, it is trivial to fulfill OpenMP memory model.

FreeScale MSC8156 does not keep cache coherence among L1 and L2 cache of different cores. It provides instructions to invalidate specified data in cache. One intuitional method to fulfill the relaxed-consistency model is invalidating data at each flush operation. However, compiler cannot always analyze what needs to invalidate accurately. Using conservative analysis result may usually cause to invalidate all shared data, which is a time consuming operation.

Because of the considerations stated above, we disable cache for shared sections to fulfill the memory model of OpenMP. Certainly this causes a decrement of performance. Extensions of distributed array and stream access can reduce the usage of shared memory, which makes the performance is acceptable under most situations. In Section V we have a measurement for performance affected by cache.

C. Transformation Strategy

OpenMP implementation method is well developed for CPU [7], [8]. Our transformation strategy inherits from implementation for CPU in many aspects, like parallel region outlining, transformation of work-sharing constructs and synchronization constructs, etc. In this chapter, we only focus on the differences and tricky points.

Example code

```
1 void foo() {
2   int a, b, c;
3   ...
4 #pragma omp parallel private(a) firstprivate(b)
5   {
6     ...
7   }
8 }
```

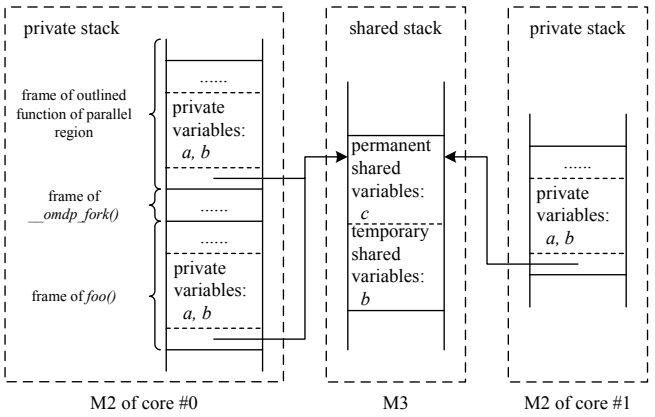

Figure 8. Illustration of shared data transformation.

1) Transformation of Shared Data: As stated in Section I, one critical problem is how to deal with shared auto variables. By default auto variables is placed on stack, which resides in core-local memory. To solve the problem, we create a *shared stack* on M3. OMDPFS puts two kinds of auto variables on shared stack:

- Auto variables shared by at least one parallel region: OMDPFS puts these variables on shared stack in its full life cycle. We call these variables as *permanent shared variables*.
- Auto variables which is not shared by any parallel region, but read or written by worker threads, for example, variables listed in *firstprivate* and *reduction* clauses. OMDPFS puts these variables on stack in core-local memory initially, and copies its value to shared stack before entering parallel region (for *firstprivate* and *reduction*) or copies the value back to private stack after exiting parallel region (for *reduction*). We call these variables as *temporary shared variables*.

Fig. 8 shows an example code and illustrates the layout of private stacks and shared stack. `a` is a private variable and only resides on private stacks. `b` appears in *firstprivate* clause, so it is a temporary shared variable, which resides in private stack of master thread initially, and OMDPFS copies its value to temporary shared frame before the parallel region. Worker threads use its value to initialize its private version. `c` is a permanent shared variable and only resides on shared stack. In master thread, shared variables are accessed by the pointer to current frame in shared stack. The frame

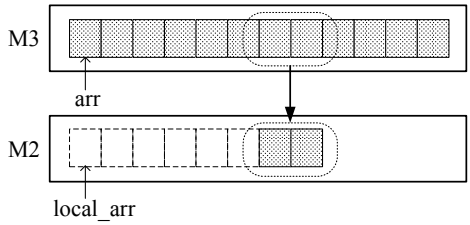

Figure 9. Illustration of transformation for distributed array.

pointer of shared stack is passed to worker threads during execution of a parallel region, so worker threads can access shared variables by that pointer.

2) Transformation of distribute and stream Directives: For each array or pointer specified in *distribute* directive, a corresponding local pointer is created and initialized by a LIBOMDP function, `__omdp_distribute()`. All references of the original array or pointer inside the statement followed by the *distribute* directive are replaced by the local pointer.

`__omdp_distribute()` allocates M2 memory for local duplication of given array. The value assigned to the local pointer is not the first address of the local duplication, but the address which pretends to be the first element of the whole array in M2. As illustrated in Fig. 9, `arr[]` is an array with 12 elements, and the main section for one thread of `arr[]` consists of `arr[6]` and `arr[7]`. The replacing pointer `local_s` is assigned by the 6^{th} elements before the allocated main section, so elements in the main section can be accessed by its original index.

Arrays and pointers specified in *stream* directives are handled in the same way as *distribute* before each chunk of the parallel loop.

3) Transformation of Data Placement Directives: FreeScale MSC8156 SDK provides "section" attribute, a kind of C extension, to map variables to specified section. In OMDPFS, we predefine three sections in our link control file, which is placed in M2, M3 and DDR respectively. OMDPC deals with shared variables on M3 or DDR and threadprivate variables on M2 by inserting corresponding attributes. For threadprivate variables on M3 or DDR, OMDPC replaces them with pointers on M2 which point to their original type. These pointers will be initialized by the global data file.

4) Content of Global Data File: Global data file contains a function `__omdp_global_initialize()`, which will be called by LIBOMDP during initialization.

LIBOMDP provides a function to allocates memory on M3 and DDR for threadprivate variables and initializes these pointers. OMDPLD generates code in the `__omdp_global_initializes()` to call this function to initialize these threadprivate variables.

OMDPC assigns one critical handle for each name of critical region and generates code to call LIBOMDP functions for *critical* directive with its critical handle. All critical handles are defined in the global data file, and `__omdp_global_initialize()` contains code to initialize these critical handles.

D. Source-to-source Compiler

We adopt Cetus [9], a source-to-source compilation framework written in Java as our infrastructure. OMDPC is built based on the C parser and intermediate representation (IR) provided by Cetus. The C parser in Cetus is based on ANTLR [10], a LL(k) parser generator. We add new kinds of IR nodes to represent OpenMDSP directives and clauses, extend the parser and write phases for OpenMDSP transformation based on strategies discussed in Section IV-C.

E. Runtime Library

We write LIBOMDP by C language with extension provided by FreeScale MSC8156 SDK. LIBOMDP depends on UniDSP for low level operations.

LIBOMDP is responsible to manage the shared stack. The shared stack is implemented as a linked list of memory blocks. Memory blocks are allocated from the heap on M3. When pushing a new frame into the stack, LIBOMDP tries to allocate space in the last node. If the available space is not enough, a new memory block is created and appended in the linked list. When a frame popping operation makes a block empty, the memory block is not freed at once. LIBOMDP merges several empty blocks to a bigger block when any empty block is to be used again. After a few operations, the linked list tends to be stable and with little fragment, and pushing and popping operations become very efficient.

V. Experimental Results

Nine benchmarks, as shown in Table I, are chosen to evaluate OMDPFS. Among them, FFT, CE, MED and DQAM are kernels in the critical stages of the LTE base station uplink, as stated in Section I-A. Normal LTE data size is chosen for the input and output dataset, both of which can be fitted into the M3 memory.

CVDP, MP and FIR are other kernels widely used in DSP. The input is designed to fit into the M3 memory. MPL and FIRL is similar with MP and FIR, but require memories beyond the M3, thus DDR is used to hold most of the data.

All benchmarks are parallelized based on legacy sequential code. For each benchmark, no more than 10 OpenMDSP directives have been used. FIR and MP utilize the extended *distribute* and *for respect* directives. For benchmarks with large data sets, FIRL and MPL utilize *stream* directive to improve the performance.

All benchmarks are compiled by OMDPFS, and optimized at O2 level using SCC. Speedups, as shown in Fig. 10, are

Table I
BENCHMARKS

ID	Application Name	Data Size
FFT	Fast Fourier Transform	signal length: 1024
CE	Channel Estimation	signal length: 1024
MED	MIMO Equalizer & Decoder	signal length: 1024
DQAM	Demodulation of 64 Quadrature Amplitude Modulation	signal length: 1024
CVDP	Complex Vector Dot Production	vector size: 4096
MP	Matrix Production	$64 \times 64 \times 64$
FIR	Finite Impulse Response Filter	signal length: 2048 response order: 256
MPL	Matrix Production (Large data set)	$512 \times 512 \times 512$
FIRL	Finite Impulse Response Filter (Large data set)	signal length: 1048576 response order: 256

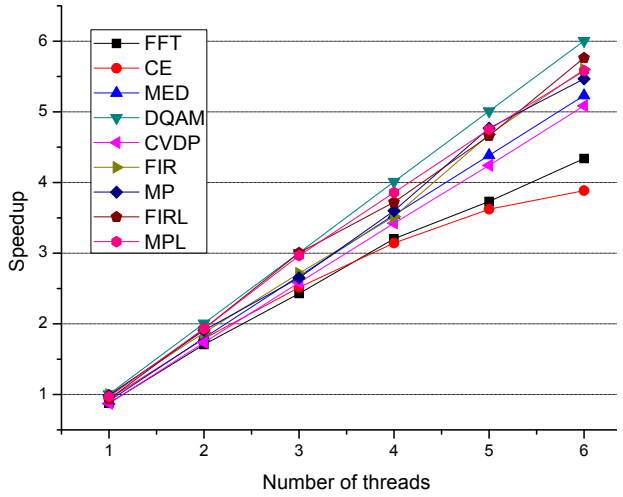

Figure 10. Speedups of each benchmark, normalized to performance of serial version with cache enabled.

derived using 1 to 6 parallel threads, normalized with the run time of the serial versions with cache enabled.

With one thread, the average speedup is 0.9, with the worse case speedup of 0.874. With six threads, 7 out of 9 benchmarks have achieved a speedup of 5+. The other two, FFT and CE, have achieved 4.34 and 3.89 speedup respectively.

The poor speedup of FFT is due to the irregular memory access pattern, which prevents it from utilizing the distributed array. CE is constituted with several small loops, each of which consumes a small portion of the total run time. The overhead of management becomes dominant when more threads are used, leading to its relatively flat speedup curve.

Additionally, we manually parallelized all these benchmarks and carefully tuned the performance to explore the potential of the parallelized code. The performance comparison between two versions is shown in Fig. 11. Using OMDPFS to compile, CE and CVDP suffer from significant overhead incurred by the runtime library, resulting in a

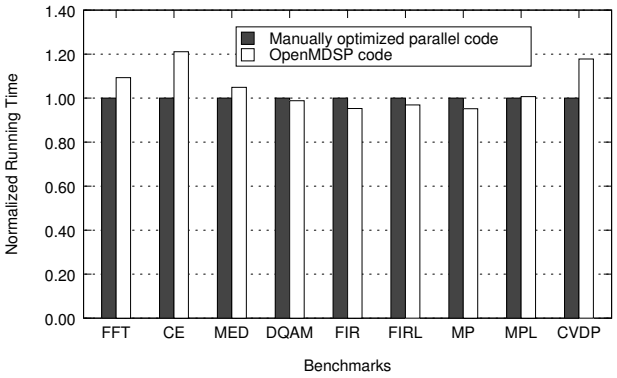

Figure 11. Comparison OpenMDSP performance with manually parallelized and optimized program running with 6 threads. Running time is normalized to running time of manually parallelized programs of each benchmark.

20% performance gap with the manual version. FFT also loses by 10% because it has a parallel inner loop, which produces noticeable loop management overhead. For some benchmarks (FIR, FIRL and MP), OMDPFS even wins the performance against its manual counterparts. This is caused by some subtle fluctuations caused by the compiler and the underlying hardware. For example, OMDPFS outlines parallel regions, which can affect register allocation.

In our benchmarks, several can use *distribute*, *for respect* and *stream* directives when writing parallel code. Fig. 12 compares the performance with and without these directives. Among all benchmarks, FIR and MP benefit the most from the distributed array because they can have excessive reuses of the array elements. In MED, each element of the distributed array is only used twice, thus it benefits less from *distribute* and *for respect* directives. In CVDP, distributed array even harms the performance because there's no reuse for each element. *stream* directive brings significant performance improvements for FIRL and MPL, which can also be credited to the array element reuses.

Overall, the extended *distribute*, *for respect* and *stream* directives are beneficial for applications with reusable array. But with little or no data reuse, they could also carry opposite effects.

VI. Limitation and Discussion

In this paper, we use extended OpenMP to express data parallelism and use vendor SDK to support task/pipeline parallelism. While this combination is a deliberate design decision whose advantage is to be able to reuse current protocol code and incremental adoption, the disadvantage of this design is that it hurts portability. It is obvious that porting the code from one vendor's DSP to another's would not be easy since the task/pipeline parallelism is written with vendor SDK's. But using OpenMP still helps portability between different generation products of the same vendor

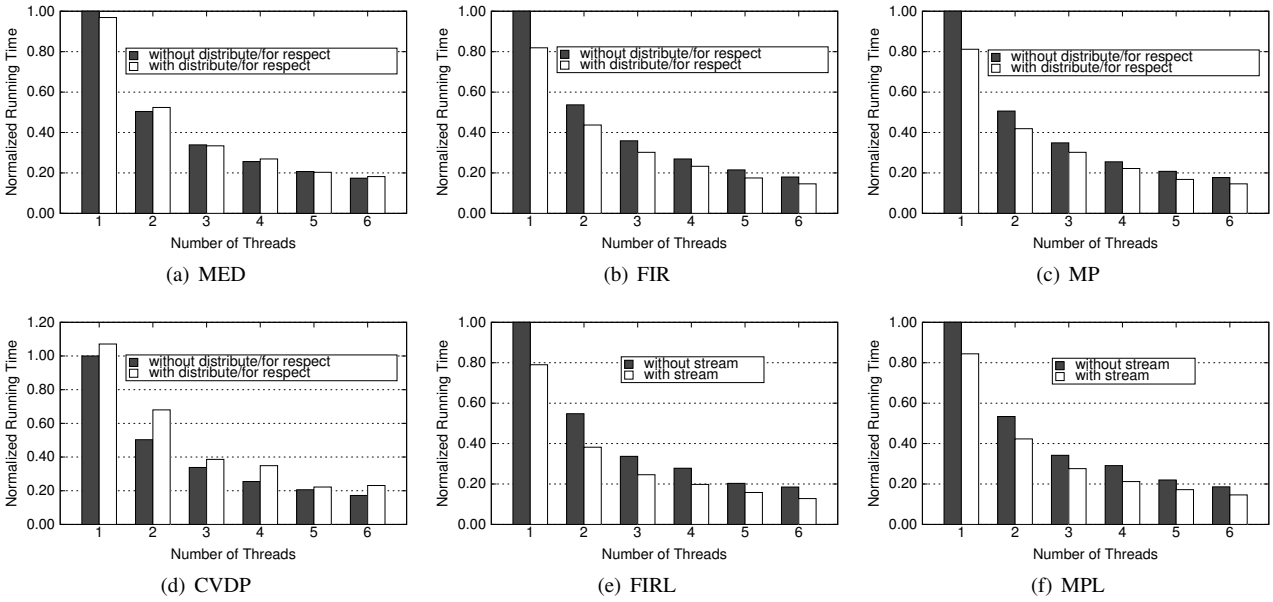

Figure 12. Comparison of running time by varying usage of distributed array or stream. Running time is normalized to one thread without using any extension.

where most vendor SDK remains unchanged and only the number of cores or size of memory hierarchy changed.

An interesting follow-up question is: Can OpenMP alone be extended to program multi-core DSPs? What are the missing features of the OpenMP programming model? Our opinion is that a complete programming model for multi-core DSP must support all three kind of parallelism: task, pipeline and data parallelism efficiently. Although OpenMP 3.0 supports task parallelism, but it still does not support pipeline parallelism well. Thus we think it is not for OpenMP to serve as a complete programming model for multi-core DSP yet.

VII. Related Work

OpenMP [5] is an industry standard parallel programming model. During the last ten years, it has been implemented in most mainstream compilers for CPU like GCC [7], Intel C Compiler [11], Open64 [8], Microsoft Visual C++. Researchers also investigated techniques for optimization OpenMP implementation on multi-core CPU and SMP [11]–[14]. OpenMP implementation technology is well developed for CPU.

OpenMP has also been implemented on some architectures other than general purpose CPUs. There are several OpenMP implementations for Cell [15], [16], GPGPUs [17], [18] and Software Scalable System on Chip(3SoC) [19], [20]. OpenMP is extended in these works to support both complex memory hierarchy and heterogeneous system issues. Regarding the memory hierarchy directives, the *data placement* directive in OpenMDSP share some common ideas with the data mapping clause in [18], but *distributed array* and *stream access* directives are unique comparing these works.

Many other programming models have also been investigated for DSP and other Multi-Processor Systems-On-Chip (MPSoCs) architecture, such as StreamIt [4], SoC-C [3], OSCAR [21], [22] and MPSoC Application Programming Studio (MAPS) [23]. We have discussed the strengths and weaknesses of StreamIt in Section I-B so we skip the discussion of it here. SoC-C [3], OSCAR and MAPS [23] are all good at task/pipeline parallelism support but lacks high level abstractions for data parallelism. We think OpenMDSP is complementary with these works to express the data parallelism inside a task.

Regarding the portability of programming model, researchers also proposed a retargetable parallel programming framework for MPSoC [24]. They designed common intermediate code (CIC) and developed a framework to map task codes to CIC. They used XML file to describe relations between tasks, and Message Passing Interface (MPI) and OpenMP can be used in a task. Currently, the framework translate OpenMP to MPI code.

VIII. Conclusions

Programming multi-core DSP systems is important yet challenging. The key problem we address in this paper is to deal with core-local memory and non-cache coherent shared memory with high-level directives. Our design and implementation show that these memory hierarchies can be managed effectively with just a few extensions to the OpenMP 2.5 standard. We expect that this work can motivate more investigations on programming these memory hierar-

chies which is critical to the success of future multi/many-core systems.

ACKNOWLEDGMENT

We thank anonymous reviewers for their suggestions. We would also thank Dehao Chen, Jidong Zhai and Tianwei Sheng for their useful comments on this paper. Finally, we would thank Ziang Hu, Qian Tan and Libin Sun for their help on experiments.

REFERENCES

[1] L. Karam, I. AlKamal, A. Gatherer, G. Frantz, D. Anderson, and B. Evans, "Trends in multicore dsp platforms," *Signal Processing Magazine, IEEE*, vol. 26, no. 6, pp. 38 –49, 2009.

[2] J. Zyren, *Overview of the 3GPP Long Term Evolution Physical Layer*, http://www.freescale.com/files/wireless_comm /doc/white_paper/3GPPEVOLUTIONWP.pdf, July 2007.

[3] A. D. Reid, K. Flautner, E. Grimley-Evans, and Y. Lin, "Soc-c: efficient programming abstractions for heterogeneous multicore systems on chip," in *CASES*, E. R. Altman, Ed. ACM, 2008, pp. 95–104.

[4] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *International Conference on Compiler Construction*, Grenoble, France, Apr 2002. [Online]. Available: http://groups.csail.mit.edu/commit/papers/02/streamit-cc.pdf

[5] OpenMP Architecture Review Board, "The openmp api specification for parallel programming," http://www.spec.org. [Online]. Available: http://www.spec.org

[6] Freescale Semiconductor, *QUICC Engine(TM) Block Reference Manual with Protocol Interworking*, Feb 2010.

[7] GCC Community, "GOMP: An openmp implementation for gcc," http://gcc.gnu.org/projects/gomp. [Online]. Available: http://gcc.gnu.org/projects/gomp

[8] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng, "Openuh: an optimizing, portable openmp compiler: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 19, pp. 2317–2332, December 2007. [Online]. Available: http://portal.acm.org/citation.cfm?id=1298358.1298361

[9] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," *Computer*, vol. 42, pp. 36–42, 2009.

[10] T. J. Parr and R. W. Quong, "Antlr: A predicated- *ll(k)* parser generator," *Softw., Pract. Exper.*, vol. 25, no. 7, pp. 789–810, 1995.

[11] X. Tian, M. Girkar, S. Shah, D. Armstrong, E. Su, and P. Petersen, "Compiler and runtime support for running openmp programs on pentium-and itanium-architectures," *High-Level Programming Models and Supportive Environments, International Workshop on*, vol. 0, p. 47, 2003.

[12] M. S. Müller, "Some simple openmp optimization techniques," in *WOMPAT*, ser. Lecture Notes in Computer Science, R. Eigenmann and M. Voss, Eds., vol. 2104. Springer, 2001, pp. 31–39.

[13] X. Tian, M. Girkar, A. J. C. Bik, and H. Saito, "Practical compiler techniques on efficient multithreaded code generation for openmp programs," *Comput. J.*, vol. 48, no. 5, pp. 588–601, 2005.

[14] B. M. Chapman and L. Huang, "Enhancing openmp and its implementation for programming multicore systems," in *PARCO*, ser. Advances in Parallel Computing, C. H. Bischof, H. M. Bücker, P. Gibbon, G. R. Joubert, T. Lippert, B. Mohr, and F. J. Peters, Eds., vol. 15. IOS Press, 2007, pp. 3–18.

[15] K. O'Brien, K. M. O'Brien, Z. Sura, T. Chen, and T. Zhang, "Supporting openmp on cell," *International Journal of Parallel Programming*, vol. 36, no. 3, pp. 289–311, 2008.

[16] H. Wei and J. Yu, "Loading openmp to cell: An effective compiler framework for heterogeneous multi-core chip," in *IWOMP*, ser. Lecture Notes in Computer Science, B. M. Chapman, W. Zheng, G. R. Gao, M. Sato, E. Ayguadé, and D. Wang, Eds., vol. 4935. Springer, 2007, pp. 129–133.

[17] S. Lee, S.-J. Min, and R. Eigenmann, "Openmp to gpgpu: a compiler framework for automatic translation and optimization," in *PPOPP*, D. A. Reed and V. Sarkar, Eds. ACM, 2009, pp. 101–110.

[18] S. Lee and R. Eigenmann, "Openmpc: Extended openmp programming and tuning for gpus," in *SC*. IEEE, 2010, pp. 1–11.

[19] F. Liu and V. Chaudhary, "Extending openmp for heterogeneous chip multiprocessors," in *ICPP*. IEEE Computer Society, 2003, pp. 161–.

[20] F. Liu and V. Chaudhary, "A practical openmp compiler for system on chips," in *WOMPAT*, ser. Lecture Notes in Computer Science, M. Voss, Ed., vol. 2716. Springer, 2003, pp. 54–68.

[21] K. Kimura, M. Mase, H. Mikami, T. Miyamoto, J. Shirako, and H. Kasahara, "Oscar api for real-time low-power multicores and its performance on multicores and smp servers," in *LCPC*, ser. Lecture Notes in Computer Science, G. R. Gao, L. L. Pollock, J. Cavazos, and X. Li, Eds., vol. 5898. Springer, 2009, pp. 188–202.

[22] A. Hayashi, Y. Wada, T. Watanabe, T. Sekiguchi, M. Mase, J. Shirako, K. Kimura, and H. Kasahara, "Parallelizing compiler framework and api for power reduction and software productivity of real-time heterogeneous multicores," in *LCPC*, ser. Lecture Notes in Computer Science, K. D. Cooper, J. M. Mellor-Crummey, and V. Sarkar, Eds., vol. 6548. Springer, 2010, pp. 184–198.

[23] R. Leupers and J. Castrillón, "Mpsoc programming using the maps compiler," in *ASP-DAC*. IEEE, 2010, pp. 897–902.

[24] S. Kwon, Y. Kim, W.-C. Jeun, S. Ha, and Y. Paek, "A retargetable parallel-programming framework for mpsoc," *ACM Trans. Design Autom. Electr. Syst.*, vol. 13, no. 3, 2008.