Automatic Data Mapping for Heterogeneous MPSoCs

Miao Wang Hongtao You Xing Jin Fengbin Qi
Jiangnan Institute of Computing Technology, China
wangmiao@nudt.edu.cn

Abstract

Heterogeneous multiprocessor system-on-a-chip (MPSoC) becomes an important platform to accelerate applications. However, programming tools and compilation techniques for memory management on MPSoCs still lag behind. This paper presents a data mapping framework to orchestrate the data movement between local memory and off-chip memory. Data alignment, data distribution, computation distribution and communication generation are employed. In contrast to other work, our method focuses on managing the layout of the program’s data to improve data locality. Experimental results show that our framework can generate efficient code for heterogeneous MPSoCs.

Categories and Subject Descriptors D.2.8 [Software]: Metrics—programming models, compilers

General Terms Design, Experimentation, Performance

Keywords Heterogeneous MPSoC, automatic data mapping, data alignment, data distribution

1. Introduction

Multiprocessor system-on-a-chip (MPSoC) are emerging as important platforms to accelerate applications. Heterogeneous MPSoC consists of a general-purpose master core and several slave cores, which can exploit the distinct features of different cores to improve the performance. Example systems include NVIDIA GeForce8 series GPUs [1] and Intel Many Integrated Core Architecture (MIC) [2]. However, the increase in the number and heterogeneity of cores makes MPSoC programming more difficult. To meet strict performance requirements, programmers must orchestrate programming models, languages and architecture features. Furthermore, managing data transfer between on-chip memory (local memory) and off-chip memory adds another level of complexity.

To address these problems, many researches [3, 4, 5] have conducted on memory optimization, task management and scheduling for MPSoCs. Other works [6, 7, 8, 9, 10] concentrates on developing tools for data management of manycore systems. Compared to previous work, we propose an automatic data mapping framework for MPSoC in an alternative way. Our target platform is a heterogeneous MPSoC, which includes a host core and a coprocessor. The host core is responsible for resource management and scheduling while the coprocessor accelerates the application. In contrast to shared memory model, we model the target MPSoC as a distributed memory system. The automatic data mapping framework focuses on managing data layout between main memory and local memory in order to reduce memory access overhead. To improve data locality, we propose a data distribution-based method relying on data alignment. In our method, the aligned data arrays are mapped onto PEs in the same way to satisfy computation need. The framework is able to determine data and computation decomposition at the same time and generate SPMD codes for data-parallel programs. We use several benchmarks to evaluate our framework, experiments show that the codes generated by our compiler can achieve high performance.

The rest of this paper is organized as follows. Section 2 describes the compilation framework. Section 3, Section 4, Section 5 and Section 6 propose the compilation techniques used in our automatic data mapping framework. Section 7 presents the experimental results. Section 8 concludes this work.

2. Compilation Framework

We model the heterogeneous MPSoC as a distributed memory system, where each heterogeneous processing element (PE) has its own memory (local memory), and all PEs are interconnected via the on-chip communication network. Using the data-parallel C codes as input, the compiler front-end translates the program into abstract syntax tree (AST) as intermediate representation. From AST, the framework analyzes the program and examines the parallel loops. Then, automatic data mapping is performed to map data and computation onto PEs. Finally, the output SPMD programs are emitted and compiled to produce binaries for the target MPSoC.

In our framework, automatic data mapping tries to enhance data locality and reduce memory access. It includes the following four phases: data alignment, data distribution, computation distribution and data transfer code generation.

3. Data Alignment

Data alignment identifies the aligned array dimensions which can be decomposed onto the same PEs. Data alignment is abstracted as the Component Affinity Graph (CAG) partition problem [11].

3.1 CAG

Given a program with \( N \) distinct arrays, the CAG for the program \( G = (V, E) \) is a weighted undirected graph, where each node \( A_i \in V \) represents the \( i^{th} \) dimension of array \( A \) in the program. Assume that \( A \) is a \( m \)-dimensional array, all dimensions of array \( A \) form a column \( C^j \) in \( G \), with \( C^j = \{A_1, A_2, ..., A_m\} \), \( C^j \subset V \). We represent the column set in \( G \) as \( C = \{C^1, C^2, ..., C^N\} \), where \( C^k (1 \leq k \leq N) \) corresponds to an array from the \( N \) distinct arrays. An edge \( (A_k, B_l) \in E \) indicates an affinity relation between the \( k^{th} \) dimension of array \( A \) and the \( l^{th} \) dimension of array \( B \). Two edges are competing edges only if they are incident on the
same node on one side while relating to two different dimensions of the same array on the other side. A competing edge is given a same node on one side while relating to two different dimensions of the same array on the other side. A competing edge is given a weight of \( e \) \((0 < e < 1)\); a non-competing edge is given a weight of \( 1 \).

A CAG can be built from the program by identifying the reference patterns between the left-hand-side (LHS) array and the right-hand-side (RHS) arrays subscripts in loop nests. Given a statement in a loop:

\[
A[i_1][i_2]...[i_m] = g(B[j_1][j_2]...[j_n]),
\]

where the statement \( mt[0][iRing] = seed \) & \( MTPgi[iRing].wmask \) and \( mt[1][iRing] = seed \) & \( MTPgi[iRing].wmask \), then the size of array \( MTPgi \) in CAG is the program whose data can fit in the local memory of heterogeneous MPSoC.

### Data distribution

Definition 1: An in-chip program is the program whose data can fit in the local memory of heterogeneous MPSoC.

Definition 2: An out-of-chip program is the program in which all the data used in the program cannot fit in the local memory of heterogeneous MPSoC.

Data distribution maps the program data onto the PE mesh. To simplify the problem, we abstract the target platform as a linear array of PEs. Assume that arrays \( A_1, A_2, ..., A_n \) of program \( P \) need to be transferred to \( m \) PEs, the memory size of local memory for each PE is \( g \) Bytes. Let \( D_i \) be the size of array \( A_i \) \((0 < i < n + 1)\) and integer \( data\_type_i \) represent data type of \( A_i \). \( P \) is an in-chip program if the following inequation is satisfied:

\[
\sum_{i=1}^{n} D_i \times (data\_type_i/8) < m \times g \tag{1}
\]

\( P \) is an out-of-chip program if:

\[
\sum_{i=1}^{n} D_i \times (data\_type_i/8) \geq m \times g \tag{2}
\]

For in-chip programs, the partitioned arrays can be directly loaded to the local memory. For out-of-chip programs, the limited local memory can’t completely keep the program arrays. The compiler decomposes the out-of-chip arrays into data tiles to fit in the local memory of each PE.

### 4.1 Determining Data Tile Size

We first consider the data tile size of arrays for in-chip program. Given an \( n \)-dimensional in-chip array \( A \), the compiler distributes \( A \) onto the 1-dimensional PE array by partitioning the \( k^{th} \) dimension of \( A \) into blocks. The size of local array allocated on each PE satisfies the memory size of the local memory. Let \( local\_A \) represent the distributed local array on each PE. \( size(A, i) \) denote the size of the \( i^{th} \) dimension of array \( A \) and \( pe\_num \) denote the available PEs. The size of the \( k^{th} \) dimension of \( local\_A \) is \([size(A, i)/pe\_num]\), and \( size(A, i) \) \((1 \leq i \leq n, i \neq k)\) for the other dimensions.
In out-of-chip program, large arrays are split into data tiles due to the limited memory size of the local memory. Each tile to be assigned should fit in the local memory. Assume that an n-dimensional out-of-chip array A is distributed onto a linear array of PEs. The memory size of the local memory for each PE is $g$ Bytes. Let an integer $data_{type}$ represent data type of array $A$, if $A$ is single-precision floating, $data_{type} = 32$; if $A$ is double-precision floating, $data_{type} = 64$. Therefore, the total size $S$ of each data tile must satisfy that:

$$S \times (data_{type}/8) < g. \quad (3)$$

### 4.2 Data partition

Our compiler performs data distribution in three steps. First, the compiler specifies the data distribution schemes for dominant arrays through program analysis. Assume that $L$ is a nested loop with loop indexes $i_1, i_2, \ldots, i_n$, and loop $i_k$ is a parallel loop. $A$ is a dominant array with the reference $A(i_1, i_2, \ldots, i_n)$ in $L$. Since loop $i_k$ is parallelized, it indicates that the corresponding $k$-th dimension of $A$ can be BLOCK distributed across the PEs. In other words, $A[i_1, \ldots, i_{k-1}, i_{k}, i_{k+1}, \ldots, i_n]$ has data distribution scheme $(\ast, \ldots, \ast, BLOCK, \ast, \ldots, \ast)$. Second, the distribution schemes of other arrays are determined according to their alignment relationship with dominant arrays. Let $B$ be an array in loop $L$ and the $m$-th dimension of array $B$ be aligned to the $k$-th dimension of array $A$ based on data alignment, which indicates that the partition of the $m$-th dimension of $B$ conforms to the partition of the $k$-th dimension of $A$. Because the data distribution scheme of $A$ is $(\ast, \ldots, \ast, BLOCK, \ast, \ldots, \ast)$, the data distribution scheme for $B$ is $(\ast, \ldots, \ast, BLOCK, \ast, \ldots, \ast)$. With data distribution schemes, the compiler finally computes the data tiles size for in-chip arrays and out-of-chip arrays, and BLOCK or CYCLIC maps arrays onto PEs. After data distribution, each PE is associated with one or a set of data tiles.

Considering the example in Figure 1 again, compiler first specifies the distribution scheme $(\ast, BLOCK)$ for dominant array $mt$. Since the alignment results in Figure 1(b) show that the first dimension of array $MT_{pgi}$ is aligned with the second dimension of $mt$, therefore, data distribution for $MT_{pgi}$ is $(BLOCK)$. Assume that the size of array $mt$ and $MT_{pgi}$ are $19 \times 1024$ and $1024 \times 1024$ respectively, thus, this program is in-chip. Let the PE number be 8. After data partition, each PE gets two local arrays $local_{mt}[10] [128]$ and $local_{MT_{pgi}}[128]$. Figure 1(c) illustrates the program’s data layout.

### 5. Computation Distribution

Computation distribution maps the computation in the program onto the target MPSoC platform through loop distribution and loop tiling.

Loop distribution is used to decompose the iteration space of loop nests across the target PEs. As shown in Figure 2(a), assume that loop $i$ can be parallelized and the PE number is 8, after loop distribution, each PE is assigned a loop block from $istart$ to $iend$ to compute the distributed local array $local_{A}$.

Loop tiling strip-mines the local computation according to the data tiles, and automatically creates blocked versions of programs. Loop tree is built to represent the relationship of loop nests before tiling. Figure 2(b) shows the loop tree for program in Figure 2(a), where $Li$ denotes the $i$-th loop and $Si$ denotes the $i$-th statement of the loop nest.

The compiler follows three steps to perform loop tiling: In the first step, the compiler traverses each loop tree node, the tile size of a loop nest can be computed based on the size of data tiles and array access information in the loop, candidate nodes for blocking are selected. In the second step, the loop nodes of loop tree are divided into disjointed groups with the criteria that the nodes in the same group construct a perfectly nested loop. In the third step, the compiler replaces the candidate blocking loop of each group with two new loops: a tiling loop and an element loop. The element loop each time operates on a data tile and executes the loop computation. The tiling loop determines the fetch order of data tiles, and the order of computation.

Figure 2 shows an example of loop tiling. Let the data distribution scheme for array $A$ and $B$ be $(BLOCK, BLOCK)$. During data distribution, the data tile size for local array $local_{A}$ and $local_{B}$ is $s \times s$. Considering the array reference of $A$ and $B$ in loop
nests $L_2$, $L_3$, $L_4$ and $L_5$, thus these four loops can be regarded as candidate blocking loops. Then the compiler divides the loop tree into three different groups \{\text{L1}\}, \{\text{L2, L3}\} and \{\text{L4, L5}\}, as shown in Figure 2(b), each group corresponds to a perfectly nested loop. Figure 2(c) illustrates the final results after loop tiling on candidate loops in each group.

6. Data transfer code generation

Given a data parallel loop nest, the primary data sets of the loop nest reside in the main memory. In in-chip program, a DMA get will be inserted before the computation if each PE needs to fetch the data tile from main memory to the local memory. Likewise, a DMA store will be inserted after the computation if each PE needs to store the data tile back from local memory to main memory. In out-of-chip problem, communication involves transferring the data tiles multiple times. Each time, a data tile is fetched from main memory to local memory, computed and then stored back. Therefore, the out-of-chip computation repeatedly executes the in-chip computation.

The compiler generates data transfer code to move data tiles between main memory and local memory for the program. DMA get will be generated to get the RHS arrays in program while DMA put will be generated to store the LHS arrays. The generated DMA operations are naively inserted before or after the corresponding loop computation. The insertion of data transfer introduces memory access overhead, which is influenced by the loop depth and the loop count. The compiler tries to minimize such cost through DMA hoisting, which moves the inserting DMA to the outermost possible loop level. During hoisting, dead codes are first eliminated. Use-Def chain is then analyzed to examine whether the DMA operation can be hoisted and determine which loop level the DMA can be inserted.

7. Experiments

The experiment was conducted on a heterogeneous MPSoC. We utilize the distributed memory model to express the target platform. We use six benchmarks to evaluate the performance: vector \text{Add}, matrix multiply \text{MM}, \text{Euler} equations, \text{Blacksholes} option pricing model, \text{Montecarlo} methods and \text{Swim-cal2} from Swim in SPEC2000.

Figure 3 shows the cycle counts of three different methods: (1) generated parallel programs using automatic data mapping, (2) generated parallel programs using shared memory model and (3) sequential codes running on host alone. From Figure 3, we can see that the parallel codes generated by the compiler based on automatic data mapping obtain the best performance, the speedup of \text{Add}, \text{Blacksholes}, \text{Euler}, \text{Montecarlo}, \text{MM} and \text{Swim-cal2} are 6.3, 2.6, 3.8, 2.0, 9.4 and 10.0 against the sequential codes, respectively. The execution time of the parallel codes with shared memory model is very poor. This is due to the fact that program data are stored in the main memory in shared memory model. Program arrays are not privatized by the compiler, each PE adopts direct memory access to get and update data during computation. Therefore, the memory bandwidth is not fully utilized, leading the low available bandwidth, which largely reduces the program performance. Compared to shared memory model, we abstract the MPSoC as a distributed memory system and use data distribution-based method to generate efficient SPMD codes. Note that, for all the benchmarks, our framework can effectively orchestrate the data transfer between main memory and local memory, enhance data locality and achieve high performance for the program.

8. Conclusion

This paper proposed an automatic data mapping method for heterogeneous MPSoCs. Compared to previous work, our framework can automatically optimize data layout between main memory and local memory. The experimental results show that our approaches could achieve good performance gain for a range of benchmarks. Future work includes overlapping computation with communication and handling data partition for complicated programs.

References