Cross-Platform Parallel Programming in Parray: A Case Study

Xiang Cui, Xiaowen Li, Yifeng Chen

Cross-Platform Parallel Programming in PARRAY: A Case Study

Xiang Cui[1 2 4], Xiaowen Li[3], and Yifeng Chen[1 2]

[1] HCST Key Lab at School of EECS, Peking University, Beijing, China
[2] State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi, China
[3] Air Defense Forces Academy, Zhengzhou, China
[4] College of Computer & Information Engineering, Henan University, Kaifeng, China

Abstract. PARRAY (or Parallelizing ARRAYs) is an extension of C language that supports system-level succinct programming for heterogeneous parallel systems. PARRAY extends mainstream C programming with novel array types. This leads to shorter, more portable and maintainable parallel codes, while the programmer still has control over performance-related features necessary for deep manual optimization. This paper uses the case study on stepwise program refinement of matrix transposition to illustrate the basic techniques of PARRAY programming.

1 Introduction

PARRAY (or Parallelizing ARRAYs) is an extension of C language that supports system-level succinct programming for heterogeneous parallel systems [1, 2]. PARRAY extends mainstream C programming with novel array types, which are then compiled to C code with machine-generated macros and vender-specific library calls. The programming style is unified for all forms of parallelism.

Matrix transposition, as a basic linear algebra algorithm, is implemented in PARRAY to demonstrate its *cross-platform* programming features. A unified PARRAY matrix-transposition code can run on hardware platforms like CPU, MIC and GPU with only memory types modified and achieve high performance.

2 Array Types of PARRAY

The following array type `A` in paged main memory has three dimensions:

```
$parray paged float[[n][n]][m] A
```

and consists of `n*n*m` elements. PARRAY supports various other memory types such as `dmem` for GPU device memory, `micmem` for MIC memory and so on. The following commands declare two type-`A` array objects `x` and `y` as pointers and allocate memory to the pointers using the corresponding library calls of the memory type. Note that the commands are the same as `$create A(x,y)` in shorthand.

```
float *x,*y;  $malloc A(x,y)
```

Unlike C language, type `A` nests its first two dimensions together, and is also a two-dimensional type. The size `$size(A_0)` of the column dimension `A_0` is `n*n`, which is split into two sub-dimensions `A_0_0` and `A_0_1` of size `n`.

PARRAY allows array dimensions to refer to existing types. The following type `B` also consists of `n*n*m` elements:

```
$parray dmem float[[#A_0_1][#A_0_0]][#A_1] B
```

but is allocated in GPU's device memory. Its row dimension `B_1` has the same offsets as `A_1` (according to dimensional reference `#A_1`), but the sub-dimensions of the column dimension are swapped. The following PARRAY command `$copy` performed by a CPU thread duplicates `n*n*m` floats at address `x` in main memory to address `y` in GPU device memory :

```
$copy A(x) to B(y).
```

If we consider every `m` adjacent elements as a unit, the layout of `y` is exactly a matrix transposition of `x`. A simple way to map the elements of an array is to use `for` command like the following code of array initialization where the pointer `y` is moved to the address of each element for processing, and `(*y)` obtains the element:

```
$for B(y) {(*y)=0;}.
```

3 Case Study

The performance of matrix transposition on different hardware platforms highly depends on the underlying architecture and requires system-level programming. Unified cross-platform programming to achieve high performance is challenging. In this case study, we illustrate a simple algorithm, a cache-friendly block-wise algorithm and a tile-buffered algorithm with different levels of performance optimization. The programming style remains tidy and unified for these algorithms.

3.1 Simple Matrix Transposition

The following code performs a square matrix transposition in memory by CPU:

```
$parray {paged double [n][n]} C
$parray {paged double [#C_1][#C_0]} D
$main{......
    $for C(x),D(y){ *y=*x;}
    ......}
```

where type `C` is declared as a `n*n` double array in main memory. Type `D` also has two dimensions referred from `C` but swaped. The `for` command makes sure the pointer `x` is moved to the address of `C`'s each element which is copied to corresponding pointer `y` whose offset is calculated according to type `D`. The square matrix is transposed as a result. By changing the memory type of `C` and `D` from `paged` to `micmem` or `dmem`, the code can be easily run on MIC or GPU.

3.2 Blocked Matrix Transposition

A more effective way to do the matrix transposition is the blocked transposition algorithm. The matrix is divided into a checkerboard of small blocks. Two blocks that are symmetrically distributed with respect to the leading diagonal are identified and their data is swapped with each other with the elements of every block also in transposed form. Data distribution is defined as follows:

```
$parray {paged double [[q][n/q]][[q][n/q]]} E
$parray {paged double [[#E_0_0][#E_1_0]][[#E_0_1][#E_1_1]]} F
$parray {paged double [[#E_1_0][#E_0_0]][[#E_1_1][#E_0_1]]} G
```

where type `E` partitions the initial square dimension of `n*n` into `(q*(n/q)) * (q*(n/q))`. `F` is declared by reordering `E`'s dimensions to represent the initial array layout as `q*q` blocks of `(n/q)*(n/q)` doubles. Compared with `F`, type `G` represents the layout after transposition. The PARRAY code is as follows:

```
$main{......
    $for F_0(x),G_0(y){
        $for F_1(x),G_1(y) { *y=*x; }}
    ......}
```

where the outer `for` command moves the pointers `x` to the beginning addresses of each block before transposition and `y` after transposition respectively; then the inner `for` command handles each block.

3.3 Buffered Matrix Transposition

For different processors, data buffer could be used to further improve performance when transposing each block. Elements in one block could be fetched into a buffer and written back to memory in a more efficient way.

With MIC, in order to get higher memory bandwidth, array accesses should be vectorized. MIC has 512-bit vector registers and every 64 doubles can be fetched into one vector register. The data buffer is defined as follows:

```
$parray {vmem double [n/q][n/q]} H
```

where vector memory type `H` has the same size with one block and is used to describe the vector register buffer. The PARRAY code is as follows:

```
$main{......
    $for F_0(x),G_0(y){
        $for F_1(x) itile H, G_1(y) otile H {
            $for H(x,y) {*y=*x;}}}
    ......}
```

where `itile/otile` clause of PARRAY is used to specify the data buffer used. Actually, the above PARRAY code can be written in a more simply way:

```
$parray {paged double [n][n]} F
$parray {paged double [#F_1][#F_0]} G
$parray {vmem double [n/q][n/q]} H
$main{......
    $for G(x) itile H, H(y) otile H{
        $for H(x,y) {*y=*x;}}
    ......}
```

where the matrix will be divided to tiles automatically when doing transposition. Similarly, with GPU, shared memory can be used to avoid the large strides of accessing device memory when doing matrix transposition.

This code is tested for various matrix sizes and achieves about 78 and 88 GB/s on MIC and Nvidia K20 GPU respectively (which are about 70% peak bandwidths of contiguous data transfer on both accelerators).

Table 1. Matrix transposition v.s. peak bandwidth of contiguous data transfer.

	Simple	Block-wise	Tile-buffered	Peak bandwidth
CPU	3.49	10.45	N/A	32.89
MIC(60 cores)	4.98	6.53	78.73	101.13
Nvidia K20 GPU	7.68	12.38	87.75	150.34

4 Conclusion

This paper uses a case study on stepwise program refinement of matrix transposition to illustrate the basic techniques of PARRAY programming and its *cross-platform* programming features. Layout patterns before and after transposition can be defined using PARRAY's *array types* easily and clearly. A unified PARRAY matrix-transposition code can run on hardware platforms like CPU, MIC and GPU with only memory types modified and achieve high performance.

5 Acknowledgement

This research is supported by the National HTRD 863 Plan of China under Grant No. 2012AA010902, 2012AA010903; the National Natural Science Foundation of China under Grant No. 61240045, 61170053, 61379048; the China Postdoctoral Science Foundation under Grant No. 2013M540821; the State Key Laboratory of Mathematical Engineering and Advanced Computing under Grant No. 2013A12; the Science and Technology Key Project of Education Department of Henan Province under Grant No. 13A520065.

References

1. Yifeng Chen and Xiang Cui and Hong Mei: ARRAY: A Unifying Array Representation for Heterogeneous Parallelism. PPoPP'12. (2012)
2. PekingUniversityManycoreSoftwareResearchGroup: http://code.google.com/p/parray-programming/. (2014)