Tiles: A New Language Mechanism for Heterogeneous Parallelism

Yifeng Chen
HCST Key Lab, School of EECS, Peking University
Beijing 100871, P.R.China
cyf@pku.edu.cn

Xiang Cui* Hong Mei
HCST Key Lab, School of EECS, Peking University
Beijing 100871, P.R.China
cuixiang08@sei.pku.edu.cn / meih@pku.edu.cn

Categories and Subject Descriptors   D.3.3 [Programming Languages]: Language Constructs and Features – Concurrent programming structures

General Terms   Languages, Performance

1. Introduction

Language mechanisms are ingredients to a programming language. A popular mechanism like object orientation often finds its way into a variety of languages. However, the existing common-purpose language mechanisms have not reflected the emerging diversity of modern heterogeneous architectures and applications.

This paper studies the essence of heterogeneity from the perspective of language mechanism design. The proposed mechanism, called tiles, is a program construct that bridges two relative levels of computation: an outer level of source data in larger, slower or more distributed memory and an inner level of data blocks in smaller, faster or more localized memory:

```
#pragma tiles loadstore(x,S,T)
{ ... user code ... }
```

where x is a pointer to an array, S and T are array types that describe the layout and distribution of source data and block-partitioned buffers respectively, and the keyword loadstore indicates both loading from and storing to the outer memory. In the user code of the program construct, the variable x points to an inner-memory data block acting like an “internal buffer”. The partitioned blocks of the source data are first copied to the inner-memory buffer of a thread, which runs the user code and then stores the data blocks back to the outer memory. The number of threads is determined automatically in runtime. Multiple heterogeneous levels are bridged by nesting of program constructs. Tiles can become pragma, types or simply a library, but being part of a language allows better type-guided performance optimization.

Tile partitioning is often the main part of performance optimization. The inner memory is usually limited in size and often requires some particular shape and data layout to reach reasonable performance. For example, multiple computing steps may compete for the limited on-chip shared memory. Optimized allocation requires semantic understanding of several computing steps within a code — a particularly difficult task to the compiler if the average cache size per core is too small to ensure performance-transparent memory accesses.

With tiles, a programmer is aware that the data need to be partitioned into blocks, but low-level coding of hardware features is not required. The size and shape of the blocks can be determined by trial and error during performance tests.

The simple pattern of tiles covers a surprisingly wide range of computation such as matrix transposition, multiplication and various other BLAS routines. Modern multicore especially manycore processors heavily rely on vectorization, but less than 30% loops in real codes like GROMACS are vectorized by the compiler. If the inner memory corresponds to vector registers instead, the tiles program construct will load (or gather) source data into vector registers; the user code is compiled into vector instructions that operate on vectors; and the result vectors will be stored or scattered back to the outer memory.

Some patterns suitable for tiles are not so obvious at the first glance. If the size of source data is too large for the device memory of GPU, then tiles becomes useful in partitioning the source data into blocks for the GPU to process (probably in a streaming mode). In this scenario, the GPU’s memory acts like a “cache” to main memory. If both outer memory and inner memory are the main memory, and the inner memory is small enough to fit in CPU’s cache, then tile loading and storing essentially performs some cache-oriented performance optimization. The idea is even applicable to message-passing parallelism: if the source data in “outer memory” are distributed over multiple processes, and the inner memory is within one process, then each block of the source data will be collected in messages passed to the process.

2. Overview

Tiles are compatible with a variety of array notations. Here we adopt a simplified version that resembles [1]. The following type defines a row-major 2D array type of 100×200 floats:

```
#pragma typedef S as float[data=100,mul=200][data=200,mul=1]
```

where S is the type name, data=n indicates that the corresponding dimension is a data dimension of length n which can be a variable whose value is determined in runtime (n=1 by default), and mul1=m denotes the offset gap between adjacent indices (m=1 by default). Other possible dimensional parameters also include add=a for starting offsets and windowing (a=0 by default), mod=r for modulo of cyclic offsets (r=0 without modulo by default), threads=n indicates a dimension in distributed memory of n threads/processes over which the data are distributed.
The index \( i \) offset of a dimension is defined as \((i^*m + a) / x\)
while the offsets of the elements reflect a row-major layout such that
S\((i,j)\) = \(i*200 + j*1\). The column-major version of the above
type can be defined:

```c
#pragma typedef R as
    float[data=100,mul=1][data=200,mul=100] .
```

Copying an array of type A to an array of type B essentially
transposes the matrix element by element. The memory in which an array
is allocated can be declared by a preceding directive:

```c
#pragma mense type micumem
where micmem (or other memory types such as host memory hmem
and SIMD vectors vmem) indicates the on-card memory of MIC
accelerators. Consider the type \( T \) of tile blocks:

```c
#pragma typedef T as
    float[data=16,mul=16][data=16,mul=1] .
```

Then the matrix-transposition code with tile partitioning becomes
a simple tiles construct with an empty body:

```c
#pragma typedef W as
    float[data=98,mul=200,add=1][data=198,mul=1,add=1] .
```

The compiler performs array transposition in the buffer as part of
the optimized storing operation. Unlike automatic parallelization
of sequential loop, tiles allow the programmer to control the
size (16x16) and shape of on-chip buffers. The number of parallel
thread created will depend on implementation. In this case,
it will be determined by OpenMP using such a straightforward
method of code generation.

Stencil computation is widely used in applications where each
mesh cell requires only a small number of instructions to compute
from the values of the neighboring cells. A typical optimization
strategy is to group multiple neighboring cells together in a tile
for each thread to increase granularity. In a typical scenario, the
updated area contains only the mesh interior, which is partitioned
into non-overlapping (smaller) blocks. The area to be loaded into
the buffers covers the entire mesh, which is partitioned into the
same number of (larger) blocks, but with some overlap. Distances
between adjacent blocks are automatically determined according
to the first block partition. Consider a simple 5-point stencil with
width-1 boundaries. The update window of interior is defined:

```c
#pragma typedef W as
    float[data=98,mul=200,add=1][data=198,mul=1,add=1] .
```

where type \( W \) describes a 98x198 window carved out from type \( S \).
Suppose the type of loaded blocks is \( T \). The vector block type \( U \) of
storing (or scattering) is slightly smaller:

```c
#pragma typedef U as
    float[data=14,mul=14][data=14,mul=1] .
```

The following figure illustrates the tile partitioning of stencil.

![Tile Partitioning](image)

The actual code for stencil computation is as follows:

```c
#pragma origin T(i,j)
#pragma tiles store(y,W,U), load(x,S,T)
for (int i=0; i<size_0(U); i++)
for (int j=0; j<size_1(U); j++)
y[U(i,j)] = (x[T(i+1,j+1)] + x[T(i,j+1)] +
```

```c
x[T(i+1,j+2)] + x[T(i+2,j+1)])/4; }
```

where size_0(U) and size_1(U) return dimension size 14. The
origin points of different buffers must be aligned correctly. Note
that tile constructs can be nested just like nested program loops.
Each construct layer then represents the bridge between two adja-
cent levels of a memory hierarchy (from hard disks all the way up
to registers).

For another example, the following two array types are distrib-
uted over 100 processes, and the communication between them es-
sentially corresponds to the Alltoall collective in MPI:

```c
#pragma typedef P as
    float[threads=100][data=100,mul=2][data=2] .
#pragma typedef Q as
    float[threads=100,mul=2][data=100] .
```

Complex distributed communication patterns must be implement-
ated with caution as it could generate a large number of fragment-
ed short messages that could dramatically reduce the bandwidth.
An optimization strategy takes advantage of the type representa-
tion and gather the scattered data to fixed-length contiguous send
buffers for different target processes. The buffering can be over-
lapped with communication. The receiving processes will scatter
the data in the receive buffers according to the type information.
The following figure illustrates the effectiveness of this approach
on 128 servers that exchange a fixed 8GB data in total.

![Communication Performance](image)

Packing/unpacking fine-grained messages no longer requires much
overhead, as the processes share the same typing information about
the overall pattern of data distribution.

3. Conclusions

As tiles are often the key insight to many non-trivial parallel algo-
rithms, its exposure to the programmer allows them to apply algo-
rithmic creativity. The new program construct is similar to Ope-
nACC, HMPP, and OpenMP Offload for MIC in that the same
variables are used for both outer and inner memory. The nest-
ing of tile constructs is similar to the programming style of Se-
quioa [2] in which different levels may reuse the same computa-
tional code. The programming style of on-demand load/store
of tiles resembles that of PGAS, but the difference is that data
layout and distribution are controlled by types. The use of prag-
ma syntax here is not attempted to ensure semantic equivalence
with pragma-free sequential code. This research is supported by
grants 863: 2012AA010902, 2012AA010903, NSFC: 61170053,
61432018, 61379048, and SKLMEAC: 2013A12.

References
