Hierarchical Array Layouts Require Domain-Specific Compiler Support

Jeyerajan Thiyagalingam, Olav Beckmann, and Paul H. J. Kelly

Department of Computing, Imperial College London,
180 Queen’s Gate, London SW7 2AZ, United Kingdom

http://www.doc.ic.ac.uk/~ob3
Introduction

Arguably, there should be programming-language support for hierarchical arrays \(^a\) \(^b\)

Or indeed, hierarchical array layout might be a better default layout than row- and column-major \(^c\)

However desirable, this has not happened yet.

Might be more useful to provide hierarchical arrays as domain-specific abstractions within existing programming languages.

The problem then becomes that we compile a domain-specific abstraction with a commodity compiler that has no knowledge of the semantics of that abstraction.

\(^a\) Wise, Frens: Morton-order matrices deserve compilers’ support.
\(^b\) Wise, Frens, Gu: Language support for Morton-order matrices.
\(^c\) Thiyagalingam, Beckmann, Kelly: Is Morton layout competitive for large two-dimensional arrays, yet?
Strength Reduction of Array Address Calculations

For an $N \times N$ row-major matrix, storage location of element $A[i][j]$ is
$S_{rm}(i, j) = N \times i + j$.

Strength-reduction: $S_{rm}(i, j + 1) = S_{rm}(i, j) + 1$.

For Morton: $S_{Morton}(i, j) = D_1(i) \mid D_0(j)$, where

- $D_0(i) = 0i_{n-1} \ldots 0i_10i_0$
- $D_1(i) = i_{n-1}0 \ldots i_10i_00$

In practice, we store these in precomputed tables.

Strength reduction of $S_{Morton}(i, j + 1)$:

\[
\begin{align*}
D_0(i + 1) &= ((D_0(i) \mid \text{Ones}_0) + 1) \& \text{Ones}_1 \\
D_1(i + 1) &= ((D_1(i) \mid \text{Ones}_1) + 1) \& \text{Ones}_0
\end{align*}
\]

where

\[
B(\text{Ones}_0) = 10101 \ldots 01010 \quad \text{and} \quad B(\text{Ones}_1) = 01010 \ldots 10101
\]

Reduces incrementing a Morton index to three arithmetic instructions.
**So what’s the problem with that?**

- With row-major strength reduction, compilers understand that neighbouring words can be packed into SIMD registers.

- However, compilers do not understand the semantics of the equivalent Morton strength-reduction:

  \[ D_0(i+1) = ((D_0(i) \mid \text{Ones}_0) + 1) \& \text{Ones}_1 \]

**Better strength-reduction:** \( S_{\text{Morton}}(i, j+k) = S_{\text{Morton}}(i, j) + D_0(k) \) iff \( \exists n \) such that \( j \mod 2^n = 0 \) and \( k < 2^n \).

- Example: Assume \( j = 4 \), then
  \[
  S_{\text{Morton}}(i, j+1) = S_{\text{Morton}}(i, j) + 1, \\
  S_{\text{Morton}}(i, j+2) = S_{\text{Morton}}(i, j) + 4, \\
  S_{\text{Morton}}(i, j+3) = S_{\text{Morton}}(i, j) + 5
  \]

- Strength-reduction of index calculation is essential for using Morton storage layout effectively.
Manual Unrolling of Morton Order Matrix Multiply

```c
void mmijk_unrolled(unsigned sz, FLOATTYPE *A, FLOATTYPE *B, FLOATTYPE *C) {
    unsigned i, j, k;
    for (i = 0; i < sz; i++) {
        unsigned int t1i = MortonTab1[i];
        for (j = 0; j < sz; j++) {
            unsigned int t0j = MortonTab0[j];
            for (k = 0; k < sz; k += 4) {
                unsigned int t0k = MortonTab0[k];
                unsigned int t1k = MortonTab1[k];
                C[t1i+t0j] += A[t1i+t0k] * B[t1k+t0j];
                C[t1i+t0j] += A[t1i+t0k+2] * B[t1k+t0j+1];
                C[t1i+t0j] += A[t1i+t0k+8] * B[t1k+t0j+4];
                C[t1i+t0j] += A[t1i+t0k+10] * B[t1k+t0j+5];
            }
        }
    }
}
```

Hierarchical Array Layouts Require Domain-Specific Compiler Support—5/13
Performance of Unrolling with Morton Address SR

MMikj on P4: Performance in MFLOP/s

- **Row-Major Default Alignment**
- **Column-Major Default Alignment**
- **Morton Default Alignment**
- **Morton 4096-Byte Aligned**
- **Morton 4096-Byte Aligned Hand-Unrolled**

MMikj on Athlon: Performance in MFLOP/s

- **Row-Major Default Alignment**
- **Column-Major Default Alignment**
- **Morton Default Alignment**
- **Morton 4096-Byte Aligned**
- **Morton 4096-Byte Aligned Hand-Unrolled**

MMikj on Alpha: Performance in MFLOP/s

- **Row-Major Default Alignment**
- **Column-Major Default Alignment**
- **Morton Default Alignment**
- **Morton 8192-Byte Aligned**
- **Morton 8192-Byte Aligned Hand-Unrolled**

- **P 4 2.0 GHz, 512 KB, icc 7.1**
- **Athlon 2100+, 512 KB, icc 7.1**
- **Alpha EV6 21264 500 MHz, 4 GB, cc v 6.0**
- **Sun UltraSparc III 750 MHz, 24GB, cc v 6.0**
Note these are non-tiled loops.

The unrolling with strength-reduction we have shown gives a significant performance improvement.

Up to a certain problem-size, Morton storage layout with unrolling and domain-specific strength reduction now matches the performance of row-major loops over row-major arrays.
How do we implement this?

- Manual implementation of such domain-specific optimisations is a bad:
  - tedious and error-prone
  - How about different degrees of unrolling for different architectures?
- Better: active libraries or hooks into back-end compilers
- **Active Libraries** are libraries that come with the means to optimise the abstractions they implement. E.g. Blitz++, FFTW
  - This will typically require metaprogramming: writing programs that generate code on the fly, at compile-time or run-time.
- Make use of back-end compilers that allow users to program “plug-in” optimisations that the compiler will apply to their code, e.g. Phoenix.
The TaskGraph Metaprogramming Library

```c
#include <stdlib.h>
#include <TaskGraph>
using namespace tg;

int main( int argc, char *argv[] ) {
    TaskGraph T;
    int b = 1;
    int c = atoi( argv[1] );
    taskgraph( T ) {
        tParameter( tVar( int, a ) );
        a = a + c;
    }
    T.compile( TaskGraph::GCC, true );
    T.execute( "a", &b, NULL );
    printf( "b = %d\n", b );
}
```

- Complete example
- Currently three backends:
  - SUIF-1
  - ROSE
  - gcc 4
- Specifies a piece of dynamically generated code.
- A parameter to the dynamic code.
- Compile the code.
- Bind a parameter and execute the code.

```
addc.cc
./addc 1 prints b = 2.
```
A TaskGraph is a data structure holding the abstract syntax tree (AST) for a piece of code.

```c
int c = atoi( argv[1] );
taskgraph( T ) {
    tParameter( tVar( int, a ) );
a = a + c;
}
```

// Generated code from addone.cc
extern void taskGraph_0(void **params) {
    int *a = *params;
    *a = *a + 1;
}

Types and special control macros determine whether we generate code or insert a value from the surrounding context.
const unsigned int k_end = _sz - mod(_sz, _unroll);

// Precomputed tables of dilated integers

// Generates a for-loop

tFor(i, 0, _sz - 1) {
  // Generates a strided for-loop
  tForStep(k, 0, k_end - 1, _unroll) {
    for(unsigned int kk = 0; kk < _unroll; ++kk) {
      C[T1[i] + T0[j]] += A[T1[i] + T0[k] + Dilate0(kk)] * B[T1[k] + T0[j] + Dilate1(kk)];
    }
  }
  for(unsigned int kk = k_end; kk < _sz; ++kk) {
    C[T1[i] + T0[j]] += A[T1[i] + T0[kk]] * B[T1[kk] + T0[j]];
  }
}
Unrolling Morton Matrix Multiply: Generated Code

extern void taskGraph_0(void **params) {

double *A = *params;
double *B = params[1];
double *C = params[2];
unsigned int *T0 = params[3];
unsigned int *T1 = params[4];
int i, j, k;
for (i = 0; i <= 502u; i++) {
    for (j = 0; j <= 502u; j++) {
        for (k = 0; k <= 499u; k += 4) {
            C[T1[i] + T0[j]] += A[T1[i] + T0[k] + 0u] * B[T1[k] + T0[j] + 0u];
            C[T1[i] + T0[j]] += A[T1[i] + T0[k] + 1u] * B[T1[k] + T0[j] + 2u];
            C[T1[i] + T0[j]] += A[T1[i] + T0[k] + 4u] * B[T1[k] + T0[j] + 8u];
            C[T1[i] + T0[j]] += A[T1[i] + T0[k] + 5u] * B[T1[k] + T0[j] + 10u];
        }
    }
    C[T1[i] + T0[j]] += A[T1[i] + T0[500u]] * B[T1[500u] + T0[j]];    
}}

Hierarchical Array Layouts Require Domain-Specific Compiler Support— 12/13
Conclusion

- Hierarchical Arrays are a very useful domain-specific abstraction.
- Unfortunately, commodity compilers do not understand the domain-specific semantics of such abstractions, resulting in suboptimal performance.

Two possible ways to overcome this problem:
- **Active libraries, based on suitable metaprogramming tools.** The idea is to implement domain-specific optimisations as late-stage code generation, when optimisation context is known.
- **Hooks into back-end compilers.**
  E.g. **Phoenix** is a .NET CLR analysis and optimisation framework from Microsoft. Designed to support user-written plug-in optimisations or ‘domain-specific optimisation features’.