Current Landscape of strictly Compiler Based Parallelization

Dominik Zosgornik

Advanced Seminar 16/17
Institute of Computer Engineering
University of Heidelberg
Outline

- Motivation
- Automatic Vectorization
- Automatic Parallelization
  - Functional languages // Haskell
  - Imperative languages // C & C++
- Conclusion
Motivation
Why automatic parallelization?

- Sequential

  ![Sequential Diagram]

  - Easy
  - Poor performance

- Explicit parallel

  ![Explicit Parallel Diagram]

  - Difficult
  - Usually best performance
  - Error prone

- Semi-explicit parallel

  ![Semi-explicit Parallel Diagram]

  - Moderate effort
  - Good performance
  - Example: OpenMP
Motivation

Why automatic parallelization?

- Implicit (automatic) parallel

  ![Diagram: Sequential Code -> Compiler -> Parallel Program]

- Easy
- Best performance
- Safe to run

Automatic parallelization is difficult:
- Example: Static compiler analysis

```c
void add(int K, float *a, float *b) {
    for (int i = 0; i < 10000; i++)
        a[i] = a[i+K] + b[i];
}
```

- $K = -1$ → Data dependencies
  *Auto-parallelization not possible*

- $K = 0$ → No data dependencies
  *Auto-parallelization possible*
Motivation

Vectorization vs. Multithreading

Vectorization
- SIMD
- Data level parallelism
- SSE, MMX, AVX, etc.

Multithreading
- MIMD
- Task level parallelism
- Multicore
Auto-Vectorization

- **Scout (C / C++)**
- **Java Vectorization Interface**
- **Summary**
Scout
Semi-automatic Vectorization

- Source-to-Source transformation tool

Sequential Code

Scout (CLI tool)

Vectorized Code

Compiler

Vectorized Program

float a[100], b[100], x;

#pragma scout loop vectorize
for (i = 0; i < 100; i++)
  if(a[i] >= 0)
    b[i] = sqrt(a[i]);
  else
    b[i] = x;

SSE2

float a[100], b[100], x;
__m128 av, bv, cv, xv, tv;
cv = _mm_set1_ps(0.0);
xv = _mm_set1_ps(x);
for (i = 0; i < 100; i += 4) {
  av = _mm_loadu_ps(a + i);
  tv = _mm_sqrt_ps(av);
  bv = _mm_blendv_ps(xv, tv, _mm_cmpge_ps(av, cv));
  _mm_storeu_ps(b + i, bv);
}
Scout

Insight into details: Residual loop computation

for (i = 0; i < S; i++)
  for (j = 0; j < T; j++)
    // do something

- GCC & clang: Scalar operations for residual values → Performance loss

Masked vectorized
- No scalar operations
- But: No support with current vector extensions

Scout: Column vectorized
- Not all extensions support gather
- SSE: Composite loads → still better performance

Auto-Vectorization

- Scout (C / C++)
- Java Vectorization Interface
- Summary
Java Vectorization Interface
(Semi-)automatic Vectorization

- Vectorization support in Java still rudimentary
- JVI extends Jitrino
- Explicit vectorization:
  - Usage of vector classes
  - Translation into HIR vector instructions
  - Translation into native code with corresponding SIMD instructions

Source: Jiutao Nie, Buqi Cheng, Shisheng Li, Ligang Wang, and Xiao-Feng Li. 2010. Vectorization for Java. [4]
Java Vectorization Interface  
(Semi-)automatic Vectorization

- Automatic vectorization
  - Loop based
  - Usage of JIT information (static analysis)

- Performance
  - Up to 55% / 107% performance gain for scimark.fft / .lu

Source: Jiutao Nie, Buqi Cheng, Shisheng Li, Ligang Wang, and Xiao-Feng Li. 2010. Vectorization for Java. [4]
Auto-Vectorization

- Scout (C / C++)
- Java Vectorization Interface
- Summary
Summary
Auto-Vectorization

- Loop based vectorization
  - Most execution time spent in loops
- Usage of static analyses in compiler
  - Dependencies, vectorizable instructions, etc.
- Scout: Advanced techniques
  - Column vectorization, register blocking
  - More extensive than GCC/clang vectorization
- Java Vectorization Interface
  - (Auto-)Vectorization integrated into JIT compiler
Auto-Parallelization

- Haskell Parallel Runtime
- Feedback Directed Implicit Parallelism
- Intel Compiler (C / C++)
- Profile driven parallelism-detection
- Conclusion
Haskell Parallel Runtime

Basics

• Purity: There are no side effects

```haskell
int global = 0, local = 0;

int not_pure(int *i, int j) {
    *i = global++;
    return j + global;
}

// Thread 1
int A = not_pure(&local, 1);
// Thread 2
int B = not_pure(&local, 1);
```

• Side effect: Access to global variable
• Side effect: Access to input variable
• A = ?, B = ?

  – Data flow dependencies
  – Unknown behavior if parallelized
Haskell Parallel Runtime
Basics

• Purity = Stateless
  – A function call with the same parameters always returns the same value

| let x = f(1) -- T1 |
| y = f(2) -- T2 |
| in y + x |

• f(1) and f(2) independent
  – No flow dependencies
  – Order of operations irrelevant
  – No data races

• Conclusion: Task / Subcomputations easy splittable into multiple threads
Haskell Parallel Runtime
Basics

• **Laziness:** Defer evaluation until needed

\[
\begin{align*}
\lambda & \text{ let } x = 1 + 2 :: \text{Int} \\
\lambda & \text{ let } z = (x,x) \\
\lambda & \text{:sprint } x \\
x & = _ \\
\lambda & \text{:sprint } z \\
z & = (_,_)
\end{align*}
\]

• **No direct evaluation of** \(x\)

• **Thunk:** Value yet to be evaluated
  – Structure in heap
    • Points to implementation if unevaluated
    • Points to result if evaluated

\[
\begin{align*}
\text{let } x & = f(1) -- \text{Thunk 1} \\
y & = f(2) -- \text{Thunk 2} \\
in & y + x
\end{align*}
\]
Haskell Parallel Runtime
Effortless explicit parallelism

\[ \lambda \text{let } x = \text{sum } [1..100000] \]
\[ \lambda \text{let } y = \text{sum } [1..100000] \]
\[ \lambda \text{x + y} \]
\[ 10000100000 \]

- Two thunks
- Single threaded evaluation

**Spark: Thunk to be evaluated parallel**

\[ \lambda \text{let } x = \text{sum } [1..100000] \]
\[ \lambda \text{let } y = \text{sum } [1..100000] \]
\[ \lambda \text{par x par y (x + y)} \]
\[ 10000100000 \]

- Spark with par
- Two thunks → two sparks
- Multi threaded evaluation
Auto-Parallelization

- Haskell Parallel Runtime
- **Feedback Directed Implicit Parallelism**
- Intel Compiler (C / C++)
- Profile driven parallelism-detection
- Conclusion
Feedback Directed Implicit Parallelism
Speculative evaluation of work

- Idea: Find long-running thunks and spark them
  - Usage of trace information → feedback to compiler
  - Recompile with modified code
  - Fully automatic approach
Feedback Directed Implicit Parallelism
Simulation vs. reality

- Simulation shows 128x speedup potential
- Actual speedup is much lower
  - Sparking overhead
  - Locking overhead


03.02.2017
Auto-Parallelization

- Haskell Parallel Runtime
- Feedback Directed Implicit Parallelism
- **Intel Compiler (C / C++)**
- Profile driven parallelism-detection
- Conclusion
Loop parallelization support

- "icc -parallel -qopt-report test.cpp"

```c
void add(int K, float *a, float *b) {
    for (int i = 0; i < 10000; i++)
        a[i] = a[i+K] + b[i];
}
```

- \(K = -1\)
  - "loop was not parallelized: existence of parallel dependence"
- \(K = 0\)
  - "loop was auto-parallelized"
- \(K = ?\)
  - "loop multiversioned …"
Auto-Parallelization

- Haskell Parallel Runtime
- Feedback Directed Implicit Parallelism
- Intel Compiler (C / C++)
- Profile driven parallelism-detection
- Conclusion
Profile driven parallelism-detection
Extended loop parallelization with OpenMP

- Idea: Combine static + dynamic information
  - Profile run with representative input data set
  - Apply OpenMP directives to loops
  - ML-based recognition of profitable loop candidates
  - Recompile with extended OpenMP directives
Profile driven parallelism-detection
Comparison

- Benchmark: Manual parallelization vs. profile-driven (Xeon 8-core)
  - Average: 96% of the performance of hand-parallelized code
  - Profile-driven approach has extended information
    - Better recognition of profitable loops

Auto-Parallelization

- Haskell Parallel Runtime
- Feedback Directed Implicit Parallelism
- Intel Compiler (C / C++)
- Profile driven parallelism-detection
- Conclusion
Conclusion

- "Programmer typically understands the code better than the compiler" [1]
  - Pure static analysis isn’t sufficient for effective parallelization
- Auto-Vectorization:
  - Common, but not everywhere
- Auto-Parallelization in Haskell
  - Language design enables data & task level parallelism
  - But: Overhead reduces maximum performance
- Auto-Parallelization in C / C++
  - Static approach with OpenMP backend already usable
  - Dynamic approach nearly achieves best performance
  - Current research: LLVM based auto-parallelizer (e.g. Polly, Apollo)
Thanks
Questions?

Polly: LLVM optimization infrastructure
Polyhedral techniques

• High-level loop and data-locality optimizer
  – Based on abstract mathematical representation

• Optimization of memory access pattern
  – Tiling, loop fusion, etc.

• OpenMP auto-parallelization

• Auto-Vectorization

• GPU code generation

• https://polly.llvm.org/
Polly: LLVM optimization infrastructure
Polyhedral techniques

Sequential

Parallel

https://polly.llvm.org/

03.02.2017
Apollo
Speculative & dynamic auto-parallelization

Programmer

Annotated source code

Apollo Static Component

Binary Executable

Apollo Runtime System

Compile-Time

Runtime

Profiling

Code-Gen

Optimized

Optimized (misprediction)

Profiling

Original

Original

Outermost loop iterations

i=0    i=7

i=8    i=37

i=38    i=137

i=138    i=237

i=238    i=245

http://apollo.gforge.inria.fr/

03.02.2017
Scout

Performance: Residual loop computation

SSE, single precision, 4 vector lanes

AVX, single precision, 8 vector lanes