Cimple: Instruction and Memory Level Parallelism DSL

Vladimir Kiriansky, Haoran Xu, Martin Rinard, Saman Amarasinghe

PACT'18

November 3, 2018
Limassol, Cyprus
Cimple Performance Gains

Performance Gains

[Haswell]
Cimple Performance Gains

- Are we teaching the wrong algorithms?

[Haswell]
Cimple Performance Gains

- Are we teaching the wrong algorithms?

Performance Gains

- Binary Search
- Binary Tree

STL

[Haswell]
Cimple Performance Gains

- Are we teaching the wrong algorithms?
- Is STL using bad implementations?
Cimple Performance Gains

- Are we teaching the wrong algorithms?
- Is STL using bad implementations?
Cimple Performance Gains

- Are we teaching the wrong algorithms?
- Is STL using bad implementations?

Performance Gains

- Binary Search
- Binary Tree
- Skip List
- Skip List Range
- SAP Hana DB
- Volt DB
- Rocks DB [Haswell]
In-Memory Databases

• Terabyte Working Sets
  - AWS 12 TB VM

• Binary Search
  SAP Hana

• Binary Tree (partitioned)
  VoltDB

• Skip List (shared)
  RocksDB, MemSQL
Cimple Performance Gains

- Are we teaching the wrong algorithms?
- Is STL using bad implementations?
- Does our programming model match hardware?
Little’s Law

\[ L = \lambda W \]

- **Arrival Rate** \( \lambda \)
- **Concurrency** \( L \)
- **Latency** \( W \)
Little’s Law

\[ L = \lambda W \]

Concurrency \( L = 8 \)

Latency \( W = 4 \)

Arrival = Departure

Bandwidth

\( \lambda = X = 2 \)
Three Improvement Paths

A) Latency

Cimple
Three Improvement Paths

A) Latency

B) Spatial locality

milk [Kiriansky et al, PACT'16]
Three Improvement Paths

A) Latency

B) Spatial locality

C) Memory Level Parallelism (MLP)
Memory Wall

• Speculative out-of-order processors: automatically discover MLP
Memory Wall

• Speculative out-of-order processors: automatically discover MLP

• Non-blocking caches - Miss Status Handling Registers (MSHRs)

• Large Instruction Windows
Memory Wall Conquered?

- Speculative out-of-order processors: automatically discover MLP

- Non-blocking caches - Miss Status Handling Registers (MSHRs) = 10 misses

- Large Instruction Windows ~200 instructions
Branch Mispredictions
Memory Wall Conquered?

Branch Misprediction
Memory Wall Conquered?

❌ Branch Misprediction
Memory Wall

- Branch Misprediction
- Re-execution of correct path of independent tasks
ILP / MLP Vicious cycle

Low
ILP

Low
MLP
ILP / MLP Vicious cycle

Branch Misprediction
Low ILP
Low MLP
High Cache Miss Penalty
Cimple Alternative

- Avoid speculation for MLP - harness Request Level Parallelism (RLP)
- Tasks pipelined on one thread
- Cooperatively context switch on likely cache miss
Outline

• **Cimple** Co-Routines Overview
• Static and Dynamic Schedulers
• Related Work
• **Cimple** DSL and Code Generation
• Performance Evaluation
• Conclusion & Q/A
Cimple Overview
Traditional
One task per thread

- Traditional dependence chain
- Request 1 executed to completion
Binary Tree

Cimple
Binary Tree
Traditional
One task per thread

- Traditional dependence chain
- When Request 1 is complete, start Request 2
Traditional
One task per thread

- Traditional dependence chain

- When Request 2 is complete, start Request 3
Traditional Limited HW Reordering

• Traditional dependence chain

• HW out-of-order execution only if predictable&short
Cimple Co-routines
Co-operative Scheduling

- Voluntary context switches after memory access
Cimple Co-routines
Co-operative Scheduling

• Voluntary context switches after memory access

• No wait for completion - assume latency is hidden
Cimple Co-routines
Co-operative Scheduling

- Voluntary context switches after memory access
- No wait for completion - assume latency is hidden
Cimple Co-routines
Co-operative Scheduling

- Voluntary context switches after memory access
- Mark explicitly with Yield
Co-routine

Yield

- Mark voluntary context switches with **Yield**
- Must fit all in instruction window
Co-routine + Prefetch

• **Prefetch** - Overlaps loads and computation

• More requests fit the instruction window
Co-routine Static Scheduling

- Execute a group at a time
- Wait until all tasks complete
Co-routine Dynamic Scheduling

• Refill one task at a time

• Refill R4 as soon as R2 completes
Co-routine
Dynamic Scheduling

• Refill one task at a time

• Refill R5 as soon as R3 completes
Co-routine
Dynamic Scheduling

- Refill one task at a time
- Refill R6 as soon as R1 completes
Static vs Dynamic

• Is Dynamic always better?
Co-routine Vectorization

• Static Scheduling
Co-routine Vectorization

- Hybrid Static+Dynamic Scheduling
Three Keys to High MLP

- Cooperative scheduling of co-routines
  **Yield** at memory requests

- Non-blocking loads overlap with computation
  **Prefetch** avoids instruction window overflow

- Branch misprediction penalty minimized
  **If/Switch** grouping, and *branchless* code
Binary Tree Lookup

```cpp
node* BinaryTree::find(node* n, Key key) {
    while (n) {
        if (n->key == key)
            return n;
        if (n->key < key)
            n = n->right;
        else
            n = n->left;
    }
    return n;
}
```
Binary Tree Hotspots
	node* BinaryTree::find(node* n, Key key) {
		while (n) {
			if (n->key == key) // 1. cache miss
				return n;
		}
		if (n->key < key)
			n = n->right;
		else
			 n = n->left;
		}

	node* root = ...; // Some initialization

Cimple
Binary Tree Hotspots

```cpp
node* BinaryTree::find(node* n, Key key) {
    while (n) {
        if (n->key == key) // 1. cache miss
            return n;
        if (n->key < key) // 2. branch
            n = n->right;  // misprediction
        else
            n = n->left;
    }
    return n;
}
```
Binary Tree Branchless

```cpp
node* BinaryTree::find(node* n, Key key) {
    while (n) {
        if (n->key == key)
            return n;
        n = n->child[n->key < key];
    }
    return n;
}
```
Today: Cimple DSL for Experts

If it is fast and ugly, they will use it and curse you; if it is slow, they will not use it.

- David Cheriton

[Jain, The Art of Computer Systems Performance Analysis]
Today: Cimple DSL for Experts

If it is fast and ugly, they will use it and curse you; if it is slow, they will not use it.
- David Cheriton

• Performance critical database indices: Replace LLVM IR builders in JIT query engines

• C++ Standard Template Library replacement
Past:
Related Work

- GP: Group prefetching - [Chen et al'04]
  manual *static scheduling* for hash-join

- AMAC: Asynchronous Memory Access Chaining
  [Kocberber et al, VLDB’15]
  manual *dynamic scheduling*
Concurrent: C++20 co_routines

- SAP Hana [Psaropoulos et al, VLDB'18]
- Microsoft SQLServer [Jonathan et al, VLDB'18] automated dynamic scheduling
- Slower than manual GP!
  Pretty front-end, high-overhead backend
- Dynamic schedule only, no vectorization
Binary Tree Lookup

define
node* BinaryTree::find(node* n, Key key) {
    while (n) {
        if (n->key == key)
            return n;
        n = n->child[n->key < key];
    }
    return n;
}
Cimple DSL: Binary Tree

6 While(n).Do(
8   If( n->key == key ).
9     Then( Return(n) ).
10    Stmt( n = n->child[n->key < key]; )
11  ).
12  Return(n);
Cimple DSL: Binary Tree

6 While(n).Do(
7     
8     If( n->key == key ).
9     Then( Return(n) ).
10     Stmt( n = n->child[n->key < key]; )
11 )
12 Return(n);
auto c = Coroutine(BST_find);
c.Result(node*).
Arg(node*, n).
Arg(KeyType, key).
Body().

    While(n).Do(
        Prefetch(n).Yield().
        If( n->key == key ).
        Then( Return(n) ).
        Stmt( n = n->child[n->key < key]; )
    ).
Return(n);
Co-routine State

Arg(node*, n).
Arg(KeyType, key).

- **Arguments, Variables**

```plaintext
1 struct Coroutine_BST_Find {
2     node* n;
3     Key_Type key;
```
Co-routine State

c.Result(node*).
Arg(node*, n).
Arg(KeyType, key).

• Result

```c
struct Coroutine_BST_Find {
  node* n;
  KeyType key;
  node* _result;
}
```
Co-routine State

- Dynamic Schedule
- Finite State Machine

```c
struct Coroutine_BST_Find {
    node* n;
    Key_Type key;
    node* _result;
    int _state = 0;
}
```

```c
Cimple
```
Dynamic Schedule: Co-routine with `switch`

```csharp
bool Step() {
    switch(_state) {
    case 0:
        return false;
    case 1:
        return true;
    case _Finished:
        return true;
    }
}
```
Dynamic Schedule: Co-routine with `switch`

```c
bool Step() {
    switch(_state) {
    case 0:
        return false;
    case 1:
        return true;
    case _Finished:
        return true;
    }
}
```

Duff's device co-routine
Scheduler Width

- **Width** high to hide latency, low to fit state in L1

```cpp
template<int Width = 48>

using Next = CoroutineState_SkipList_next_limit;
SimplestScheduler<Width, Next>(len,
    [&](Next* cs, size_t i) {
        *cs = Next(&answers[i], IterateLimit,
                   iter[i]);
    });
```
Static Schedule: for

Vectorization friendly Struct-of-Arrays

```c
bool SuperStep() {
    for(int _i = 0; _i < _Width; _i++) {
        KeyType& k = _soa_k[_i];
        HashType& hash = _soa_hash[_i];
    }
}
```
Static Schedule: \texttt{for}

Vectorization friendly Struct-of-Arrays

```c
bool SuperStep() {
    for (int _i = 0; _i < _Width; _i++) {
        KeyType& k = _soa_k[_i];
        HashType& hash = _soa_hash[_i];
    }

    for (int _i = 0; _i < _Width; _i++) {
        KeyType& k = _soa_k[_i];
        HashType& hash = _soa_hash[_i];
    }
}
```
Applications
Binary Search

```c
Arg(ResultIndex*, result).
Arg(KeyType, k).
Arg(Index, l).
Arg(Index, r).
Body().
While( l != r ).Do(
    Stmts(R"""" { 
        int mid = (l+r)/2;
        bool less = (a[mid] < k);
        l = less ? (mid+1) : l;
        r = less ? r : mid;
    } )"""").
    Prefetch(&a[(l+r)/2]).Yield()
).
Stmt( *result = l; );
```
Binary Search

```
Arg(ResultIndex*, result).
Arg(KeyType, k).
Arg(Index, l).
Arg(Index, r).
Body().
While( l != r ).Do(
  Stmts(R""")( {
    int mid = (l+r)/2;
    bool less = (a[mid] < k);
    l = less ? (mid+1) : l;
    r = less ? r : mid;
  } )"").
  Prefetch(&a[(l+r)/2]).Yield()
).
Stmt( *result = l; );
```
Binary Search

```c
Arg(ResultIndex*, result).
Arg(KeyType, k).
Arg(Index, l).
Arg(Index, r).
Body().
While ( l != r ).Do(
    Stmts(R""") ( {
        int mid = (l+r)/2;
        bool less = (a[mid] < k);
        l = less ? (mid+1) : l;
        r = less ? l : mid;
    } )"" ).
Prefetch(&a[(l+r)/2]).Yield();
Stmt( *result = l; );
```
Cimple

Skip List Lookup

```cpp
VariableInit(SkipListNode*, n, {}). 
VariableInit(uint8, ht, {pred->height}). 
While (true).Do(
    While (ht > 0).Do ( // down 
        Stmt ( n = pred->skip[ht - 1]; ). 
        Prefetch(n).Yield(). 
        If (!less(k, n->key)).Then (Break()). 
        Stmt ( --ht; )
    ). 
    If (ht == 0).Then ( Return (nullptr ) ). 
    Stmt ( --ht; ). 
    While (greater(k, n->key)).Do ( 
        Stmt ( pred = n; n = n->skip[ht]; ). 
        Prefetch(n).Yield(). 
    ). 
    If (!less(k, n->key)).Then ( 
        Return ( n ) ); 
)```

Cimple
Skip List Lookup

```c
VariableInit(SkipListNode*, n, {}).
VariableInit(uint8, ht, {pred->height}).
While(true).Do(
  While(ht > 0).Do( // down
    Stmt( n = pred->skip[ht - 1]; ).
    Prefetch(n).Yield().
    If(!less(k, n->key)).Then(Break()).
    Stmt( --ht; )
  ).
  If (ht == 0).Then( Return( nullptr )).
  Stmt( --ht; ).
  While (greater(k, n->key)).Do(
    Stmt( pred = n; n = n->skip[ht]; ).
    Prefetch(n).Yield().
  ).
  If(!less(k, n->key)).Then(
    Return( n ));
```
Skip List Iteration

```c
While( limit-- ).Do(
    Prefetch(n).Yield().
    Stmt( n = n->skip[0]; )
).
Prefetch(n).Yield().
Return( n->key );
```

- Pointer chasing
Hash Table Lookup (Linear Probing)

- SIMD
- One cache line

```c
Result(KeyValue*).
Arg(KeyType, k).
Variable(HashTable, hash).
Body().
Stmt ( hash = Murmur3::fmix(k); ).
Stmt ( hash &amp;= this-&gt;size_1; ).Yield().
Prefetch( &amp;ht[hash] ).Yield()
&lt;&lt; R""
while (ht[hash].key &gt;= k &amp;&amp;
       ht[hash].key != 0) {
    hash++;
    if (hash == size) hash = 0;
} ""
Return( &amp;ht[hash] );
```
Performance Evaluation
Performance

- **Performance Gains**

  - Binary Search
  - Binary Tree
  - Skip List
  - Skip List Range
  - Hash Table

[Default indices of VoltDB/RocksDB]
Performance

[Default indices of VoltDB/RocksDB]
Thread Level Parallelism

- Multi-core
- SMT hardware thread
- Single-thread OS context switching?
  - 50x slower
Cimple Throughput Gains on Multicore

![Graph showing throughput gains on multicore processors]

- Baseline
- Cimple

**Throughput (MOps/s)**

- **Cores**: 1, 2, 4, 8, 12, 16, 20, 24
- **Baseline**
  - 6.4x
  - 3.7x
- **Cimple**
  - 6.4x
  - 3.7x

**[Haswell (2x12 cores)]**
Cimple Throughput Gains vs Hyper-threading

Baseline
Cimple

Throughput (MOps/s)

HW Threads

[Haswell (2x12 cores, 2-way SMT)]

6.4x

2.4x
Cimple Throughput Gains vs Hyper-threading

Throughput (MOps/s)

```
 baseline
  Cimple
```

6.4x

6x

[Haswell (2x12 cores, 2-way SMT)]
IPC Analyzed

- **SkipList Range:** scheduler overhead
  
  \[ p = p->next; \]

- **HashTable:** SIMD vectorization

![Graph showing speedup and IPC gain for different data structures: Binary Search, Binary Tree, Skip List, Skip List Range, and Hash Table. The Skip List Range shows a notable 22x increase.]
ILP Analyzed

- Uncovered more parallelism
- Absorbed scheduler overhead
MLP Analyzed

- HashTable - OoO HW extracts MLP
MLP Analyzed

- HashTable - OoO HW extracts MLP
- SkipList - speedup matching MLP gains
MLP: Ineffective or Low Performance Gains

- MLP Base
- MLP Cimple
- Speedup
- MLP Gain

- Binary Search

- Performance Gains

Cimple
MLP: Ineffective or Low Performance Gains

- MLP Base
- MLP Cimple

- Binary Search
- Binary Tree
- Skip List
- Range

- Speedup
- MLP Gain

Graph showing performance gains for different data structures.
MLP Improvement Paths

• Increased Efficiency
  - Static scheduling
  - Vectorization

• Increased Effectiveness
  - Dynamic scheduling: no bubbles
  - Branchless code
Conclusion

- Fast
  - up to 6.4× speedup

- Portable DSL (*for Stephanies*)
  - template libraries
  - database query engines

- Next: C++ standards (*for Joes*)
Thanks

Vladimir Kiriansky
vlk@csail.mit.edu