AutoTuneTMP: Auto-Tuning in C++ With Runtime Template Metaprogramming

David Pfander, Malte Brunn, Dirk Pflüger
University of Stuttgart, Germany

May 25, 2018
Vancouver, Canada, iWAPT18

SimTech

Auto-Tuning for Node-Level Optimization

Naive matrix-matrix multiplication in C++

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    for (int k = 0; k < N; k++)
      C[i * N + j] += A[i * N + k] * B[k * N + j];
```

- On 2xXeon E5-2620, 12C 2.0Ghz, N=2048, gcc 5 with ”-O3 -mavx -fopenmp”

Auto-Tuning for Node-Level Optimization

Naive matrix-matrix multiplication in C++

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    for (int k = 0; k < N; k++)
      C[i * N + j] += A[i * N + k] * B[k * N + j];
```

- On 2xXeon E5-2620, 12C 2.0Ghz, N=2048, gcc 5 with ”-O3 -mavx -fopenmp”
- 1.8 GFlops, $<$ 1% peak performance (192 GFlops)

Needed (low-level) optimizations

parallelization ✓, vectorization, tiling, software pipelining, conflict-misses, register blocking, instruction order, alignment, NUMA-aware

- Our goal: partially automate node-level performance optimization

Agenda

C++ Template Metaprogramming

AutoTuneTMP: CPPJIT, Auto-Tuner, Optimization Templates

Case Study: Matrix Multiplication

Conclusion

C++ Template Metaprogramming

Genericity	Compile-time evaluation

Genericity

```
template <typename T>
class container {
  void push_back(T &element) {
    // add element at the back
  }
};
```

Compile-time evaluation

```
constexpr int pow(int base, int exp) {
  if (exp == 0)
    return 1;
  else
    return base * pow(base, exp - 1);
}
```

- Genericity, but also compile-time code evaluation

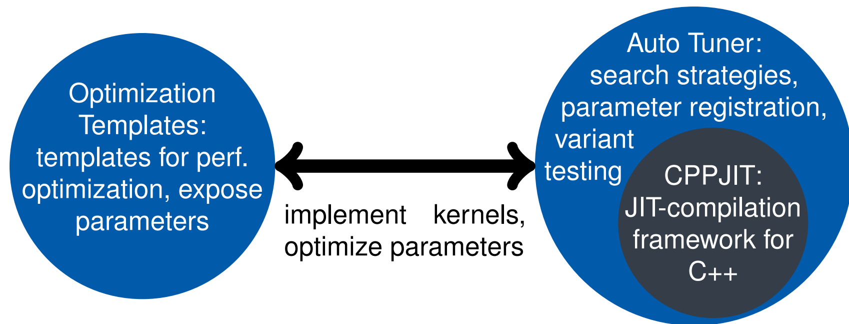
C++ Template Metaprogramming

Function type analysis

```
template <class... Args> struct pack;
template <class R, class... Args>
struct function_traits<std::function<R(Args...)>> {
  using return_type = R;              // return type
  using args_type = pack<Args...>;    // argument types
};

// usage: declare var to match return type
std::function<int(int, int)> f;
function_traits<decltype(f)>::return_type var;
```

- “Extends” language (without changes to the compiler)
- Performed in standard compiler, no runtime overhead
- We use it to (1) generate tuning API (2) perform code transformations

AutoTuneTMP Components



- C++17 framework
- Optimization templates support writing parameterized code
- Vision: optimization templates as kernel building blocks
- Library approach (requires standard C++ compiler at runtime)

CPPJIT

Declaration and compilation

```
/* global scope */
AUTOTUNE_KERNEL(
  vector<int>(vector<int> &),
  square, "kernel_src_dir")
/* local scope */
// JIT compile and run
vector<int> r = autotune::square(o);
```

JIT-compiled kernel in `square.cpp`

```
AUTOTUNE_EXPORT
vector<int> square(vector<int> &o) {
  ...
}
```

- Backend for JIT compilation is configurable: gcc, clang, nvcc, ...
- Near-seamless integration into existing code, full C++ support in kernel

CPPJIT

Declaration and compilation

```
/* global scope */
AUTOTUNE_KERNEL(
  vector<int>(vector<int> &),
  square, "kernel_src_dir")
/* local scope */
// JIT compile and run
vector<int> r = autotune::square(o);
```

JIT-compiled kernel in `square.cpp`

```
AUTOTUNE_EXPORT
vector<int> square(vector<int> &o) {
    ...
}
```

- Backend for JIT compilation is configurable: gcc, clang, nvcc, ...
- Near-seamless integration into existing code, full C++ support in kernel

Why JIT for auto-tuning?

- Create compute kernels variants as needed
- No (whole) application recompiles/restarts

Auto-Tuning With AutoTuneTMP

Auto-tuning a matrix-vector multiplication kernel with a single parameter

```
AUTOTUNE_KERNEL(vector<double>(vector<double> &, vector<double> &),
                mv_kernel, "kernel_src_dir")
```

Auto-Tuning With AutoTuneTMP

Auto-tuning a matrix-vector multiplication kernel with a single parameter

```
AUTOTUNE_KERNEL(vector<double>(vector<double> &, vector<double> &),
                mv_kernel, "kernel_src_dir")

/* local scope */
countable_set parameters;
parameters.emplace<exponential_parameter>("BLOCKING", ...);
```

Auto-Tuning With AutoTuneTMP

Auto-tuning a matrix-vector multiplication kernel with a single parameter

```
AUTOTUNE_KERNEL(vector<double>(vector<double> &, vector<double> &),
                mv_kernel, "kernel_src_dir")

/* local scope */
countable_set parameters;
parameters.emplace<exponential_parameter>("BLOCKING", ...);

tuners::bruteforce tuner(mv_kernel, parameters);
countable_set optimal = tuner.tune(m, v); //tune, specify benchmark
```

Auto-Tuning With AutoTuneTMP

Auto-tuning a matrix-vector multiplication kernel with a single parameter

```
AUTOTUNE_KERNEL(vector<double>(vector<double> &, vector<double> &),
                mv_kernel, "kernel_src_dir")

/* local scope */
countable_set parameters;
parameters.emplace<exponential_parameter>("BLOCKING", ...);

tuners::bruteforce tuner(mv_kernel, parameters);
countable_set optimal = tuner.tune(m, v); //tune, specify benchmark

mv_kernel.set_parameter_values(optimal);
vector<double> result = mv_kernel(m, v);
```

- `tune` arguments have to be reusable and copyable

Tunable AutoTuneTMP Kernel

Auto-tuning a matrix-vector multiplication kernel with a single parameter

```
#include "autotune_kernel.hpp"

AUTOTUNE_EXPORT vector<double> mv_kernel(
  vector<double> &m, vector<double> &v) {
  loop_exchange<BLOCKING>(/* ... */);
  return result;
}
```

- Uses parameterized template for performance optimization
- Parameters currently added as #define with specified value
- Parameters header created before JIT compilation

Search Strategies and Parameters

- Search strategies:
  - Brute force search, Monte Carlo search
  - Line search, parallel line search
  - Neighborhood search, full neighborhood search
- Parameters and tuners designed for extensibility

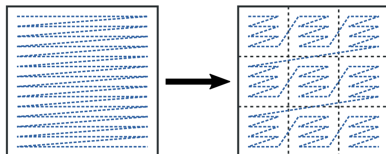
Parameter interface used by the `randomizable_set` class

```
class randomizable_parameter {
 public:
  virtual const std::string &get_name() const = 0;
  virtual void set_random_value() = 0;
  virtual const std::string get_value() const = 0;
};
```

Array Tiling

Tiling of a matrix into 2x4 blocks

```
vector<double> m = ...;

tiling_configuration conf = {{2, N}, {4, N}};
auto tiled = make_tiled<2>(m, conf);

iterate_tiles<2>(tiled, conf,
  [](tile_view<2> &view) {
  ...
});

m = undo_tiling<2>(tiled, conf);
```



- Optimization template for classical cache optimization

Register Blocking

Register-blocked inner product

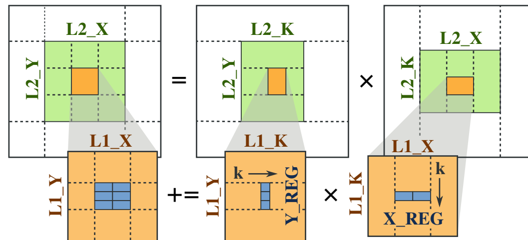
```
// with blocking factor 4 (HSW: 4*4=16 wide reg_array)
using reg_array = register_array<double_v, 4>;

reg_array r = 0.0;
for (int i = 0; i < ...; i += width) {
  reg_array a_reg(A.data() + ..., flags::vector_aligned);
  reg_array b_reg(B.data() + ..., flags::vector_aligned);
  r += a_reg * b_reg;
}
```

- For software pipelining/more instruction-level parallelism (and register reuse):

```
fmadd231pd %ymm8, %ymm9, %ymm1
fmadd231pd %ymm11, %ymm10, %ymm2
fmadd231pd %ymm6, %ymm7, %ymm0
fmadd231pd %ymm4, %ymm5, %ymm3
```

- Still close to scalar code!

(Yet Another) Auto-Tuned Matrix Multiplication



- Hardware features addressed:
  - L1 cache blocking (array tiling template, tile view template)
  - Efficient prefetching, less TLB pressure (array tiling template, tile view template)
  - Register blocking (register blocking template)
  - Software pipelining (register blocking template)
- Vc for vectorization, OpenMP for parallelization

Register-Block-Level Matrix Multiplication

```
using Vc::double_v;
using reg_row = register_array<double_v, X_REG / double_v::size()>;

// 2d register block
array<reg_row, Y_REG> acc;

for (size_t k = 0; k < L1_K; k += 1) {
  array<double_v, Y_REG> a_reg;
  for (size_t r = 0; r < Y_REG; r++)
    a_reg[r] = A_trans_view[...];
  reg_array b_reg(B_view.pointer(...), flags::vector_aligned);
  for (size_t r = 0; r < Y_REG; r++)
    acc[r] += a_reg[r] * b_reg; // FMA
}
```

- Optimized implementation fits on single slide (register blocked, tile view, vectorized)

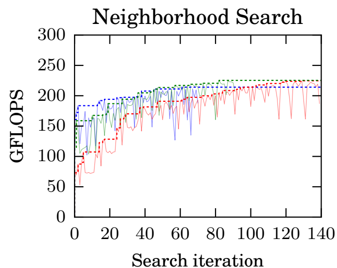
Parameters and Evaluation Scenario

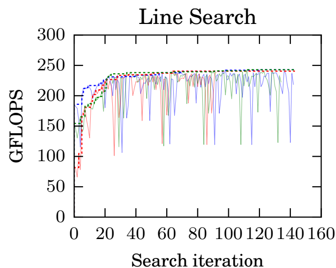
	X_REG	Y_REG	L1_X	L1_Y	L1_K	L2_X	L2_Y	L2_K	SMT
min max step	8 40 8	1 5 1	10 40 5	8 64 8	16 128 16	20 100 10	16 128 16	32 256 32	{4-way, 2-way, off}

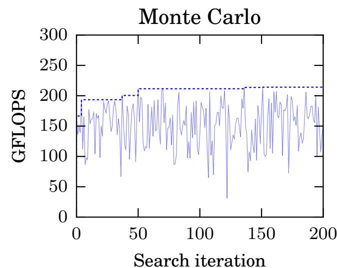
- 9 parameters, 19 353 600 valid combinations
- Scenario:
  - Square matrices sized 4096 $\times$ 4096, 5 repetitions
  - Random initial parameter combinations!
- Search strategies: line search, neighborhood search, Monte Carlo search
- Double precision arithmetic

Performance on Xeon Silver 4116

- Intel Xeon Silver 4116, 12C, 2.1 GHz base, 1.4 GHz heavy AVX512



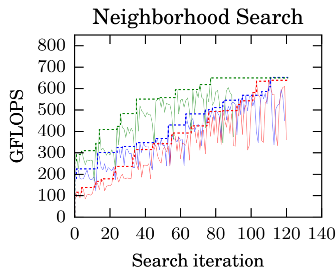


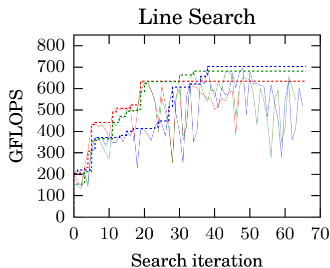


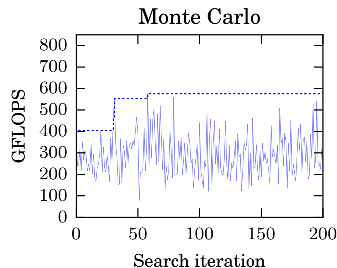
- Performance saturates within $<$100 search steps (from random start)
- Directed tuning is beneficial
- 243 GFlops, 90% peak performance! (269 GFlops peak)

Performance on Xeon Phi 7210

- Intel Xeon Phi 7210, 64C, 1.3 GHz base, 1.0 GHz heavy AVX512

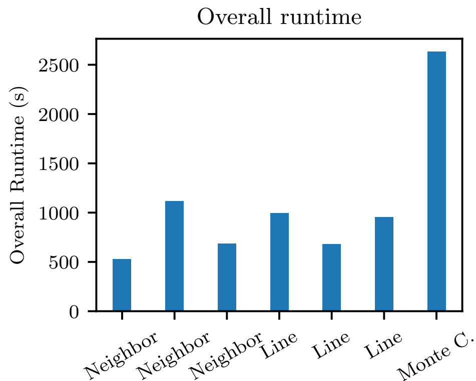


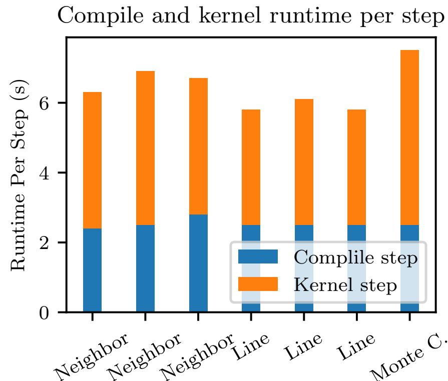




- Again, performance saturates within $<$100 search steps (from random start)
- Again, directed tuning is beneficial
- 704 GFlops, 34% peak (2048 GFlops peak)

Tuning Duration

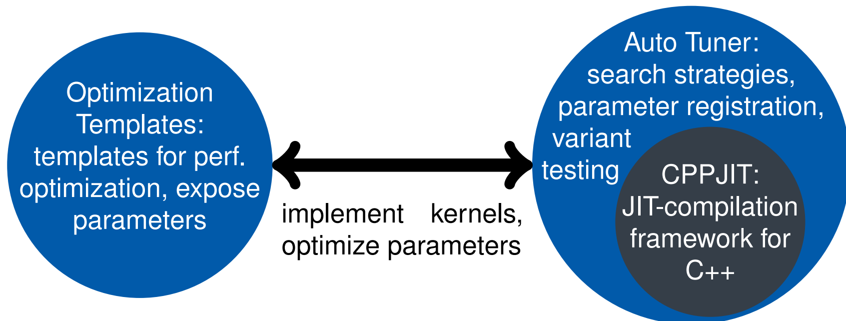




- Monte Carlo: fixed number of iterations
- Trade-off: duration vs. quality; the initial guess matters
- Parallel line search compile 0.9s (avr.)

Conclusion

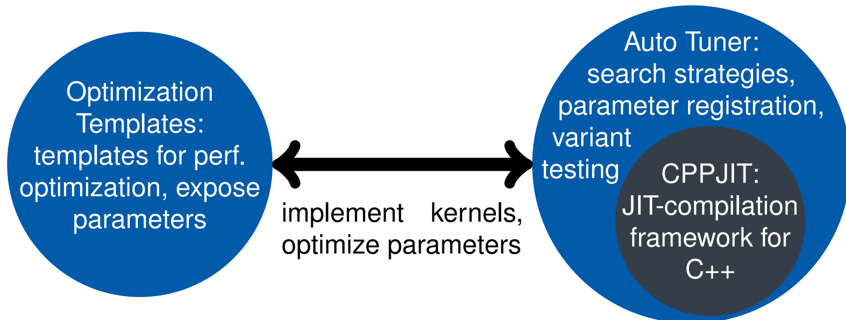
- AutoTuneTMP is an extensible, easy-to-use, easy-to-integrate auto-tuner
- Provides approach for developing tunable compute kernels



- First results for matrix multiplication demonstrate applicability and performance

Conclusion

- AutoTuneTMP is an extensible, easy-to-use, easy-to-integrate auto-tuner
- Provides approach for developing tunable compute kernels



- First results for matrix multiplication demonstrate applicability and performance

Questions?

Optimal Parameters

- Xeon Silver 4116:

Strategy	X_REG	Y_REG	L1_X	L1_Y	L1_K	L2_X	L2_Y	L2_K	SMT	GFlops
Neighbor S.	8	1	20	64	16	40	128	128	off	225
Line S.	32	5	25	32	128	100	128	128	2-way	243
Monte Carlo	40	4	32	40	96	96	80	96	2-way	218

- Xeon Phi 7210:

Strategy	X_REG	Y_REG	L1_X	L1_Y	L1_K	L2_X	L2_Y	L2_K	SMT	GFlops
Neighbor S.	24	4	28	24	128	84	48	128	4-way	653
Line S.	16	5	30	32	32	60	64	64	4-way	704
Monte Carlo	40	4	24	40	64	72	120	64	4-way	621

Why JIT compilation?

- Requirements may include the following:

Feature	External tuner	Compiler (DSL)	JIT
Large parameter space	++	?	++
Tuning speed	-	?	+
Tuning integration	+	?	++
Maintainability/Extensibility	+	++/-	++
Online tuning	-	?	+
Application domain	?	++	?

- Create kernel variants as needed!
- No application recompiles/restarts during tuning
- Key to integrated library approach (C++ & JIT)

Optimization Templates Overview

Name	Description	Impl.
tiling (logical)	iterate loops in tiles, no change of the data structure	✓
tiling	tiled iteration with tiled data structure	✓
loop unrolling	options are: fully, partial, none	✓
register blocking	to control instruction-level parallelism	✓
struct-of-array	supports vectorization	✓
vector ins. with configurable width	adapt vector width	via library (Vc)
STL Algorithms	widely used, many parameters possible	✗
padding (configurable)	to reduce cache conflict misses of large arrays	✗
loop parallelization	configurable parallelization of (nested) loop	✗

- Planed and implemented parameters, more to come (for CUDA/OpenCL C++/SyCL)
- Templates generally composable, error checking through standard compiler

Loop Exchange Template

Loop exchange template

```
loop_exchange<BLOCKING>({0, N}, {0, N},
  [&](size_t i, size_t j) {
    result[i] += m[i * N + j] * v[j];
});
```

- Partially exchanges two loops
- Exposes tunable BLOCKING parameter

Generated code for BLOCKING==5

```
for (size_t ii = 0; ii < N; ii += 5)
  for (size_t j = 0; j < N; j++)
    for (size_t i = ii; i < ii + 5; i++)
      result[i] += m[i * N + j] * v[j];
```