Master's Thesis

Improving Single Core Performance in OPAL

Lionel Miserez

Supervisors:
Prof. Dr. Peter Arbenz
Dr. Andreas Adelmann

Submitted to:
ETH Zurich
Department of Computer Science

April 9, 2016

Abstract

We improve single core performance of the `RingCyclotron` simulation of `OPAL` by applying two different approaches to its top hotspots `ParallelCyclotronTracker::derivate` and `Cyclotron::apply` and the calling context thereof, `ParallelCyclotronTracker::rk4`.

Firstly, we apply an array of aggressive optimization techniques meant to improve performance of compute-bound problems by explicitly preventing the compiler from conservatively disabling optimizations and then vectorize the obtained code using compiler `intrinsics`. Secondly, we select the optimizations that are the least intrusive in terms of reducing the readability of the code.

We contrast these two approaches in light of the speedups they yield in the relevant functions (2.66× and 1.79×) as well as their overall speedups (1.55× and 1.29×), which are limited by the fact that `ParallelCyclotronTracker::rk4` only accounts for approximately 35% of the total runtime.

Acknowledgements

I wish to acknowledge Prof. Dr. Peter Arbenz of ETH and Dr. Andreas Adelmann of PSI for their supervision and support of this master's thesis and the valuable insights, discussions and corrections they provided along the way. Furthermore, I wish to thank Uldis Locāns of PSI for his support regarding the computing resources and for sharing his knowledge of the `OPAL` library.

Table of Contents

Chapter 1
Introduction

1.1 OPAL

The Object Oriented Particle Accelerator Library (`OPAL`) is a "tool for charged-particle optics in accelerator structures and beam lines"[1]. It is provided as an extendable "open source C++ framework"[2] providing functionality for "general particle accelerator simulations including 3D space charge, short range wake fields and particle matter interaction"[2].

From a programmer's point of view the framework is fairly extensive as it offers many functionalities to be mixed and matched leading to a large codebase with many specialized functions and code fragments.

1.2 Task

This master's thesis attempts to tackle the task of improving the single core performance of the cyclotron simulation setup of `OPAL` primarily by applying vectorization. The first task is therefore to properly benchmark the application and to identify the parts of the code which are likely to gain the most from optimizing. Thereafter, the optimizations and vectorization should be applied to the already parallelized code to yield a mixed `MPI` & `AVX` implementation.

Chapter 2
Preparations

2.1 Problem Selection

Since `OPAL` is an extensive collection of functionalities, we decide to focus only on the performance of one specific application, namely the `RingCyclotron` example. Focusing only on this one application is supposed to provide the basis for an evaluation of which techniques to apply where, how to do so and whether the speedup one can expect is worth the trade-off in maintainability and readability. Optimizing the heavily used parts of that computation might also yield improvements for other applications that use the same code which would be a desirable side effect.

2.2 Identifying Areas of Work

Before attempting to improve the performance of anything it is important to have a way of verifying that the changes made do not break the code and that it still gives the same results. Furthermore, one must figure out which areas of the code are the most likely to yield decent speedups compared to the amount of work required.

The first issue is addressed by using the preexisting regression test suit for `OPAL` while in order to investigate the second point we use `Intel`'s `VTune Amplifier`[1] to profile the application, identify potential hotspots[2] in the code and track their improvements across optimization steps.

2.2.1 Profiling

In the following section we present an overview of the distribution of computational effort to compute the `RingCyclotron` example[3,4] tracking 10^6 particles. The application is obtained by compiling the baseline code[5] with `icc`[6] and optimization flags `-O3 -march=native`.[7]

2.2.1.1 Hotspot Analysis

Table 2.1 shows the result of a hotspot analysis of the `RingCyclotron` example. One observes that only the top two of the five top hotspots are not library calls related to `MPI` overhead. Combining the times for the two highlighted

[1] See Appendix A for convenient commands to invoke `VTune Amplifier` and extract the relevant information.
[2] Note that 'hotspot' in this context denotes a function which is responsible for a large amount of the runtime when not counting the time spent in its subroutines. This might be due to very costly computation or lots of invocations. Either way optimizing these functions should yield the highest payoff.
[3] See Appendix B for the input file.
[4] See Appendix C for information on the machine used.
[5] See Section 3.1.
[6] It should be noted that if we redo the analysis on code compiled with `gcc` the numbers change somewhat – in particular `ParallelCyclotronTracker::rk4` and its subroutines account for $\approx$ 5% more of the roughly 5% shorter time – but not to the extent of changing the conclusion.
[7] The `-march` flag allows setting the architecture for which the code is to be compiled. Setting this flag to `native` "selects the CPU to tune for at compilation time by determining the processor type of the compiling machine. Using `-mtune=native` will produce code optimized for the local machine under the constraints of the selected instruction set. Using `-march=native` will enable all instruction subsets supported by the local machine"[3]. Note that "`-march=cpu-type` implies `-mtune=cpu-type`"[3].

Function	CPU Time	Relative CPU Time
Cyclotron::apply	153.173s	≈ 11.2%
ParallelCyclotronTracker::derivate	129.992s	≈ 9.6%
mca_btl_vader_component_progress	118.160s	≈ 8.6%
mca_pml_ob1_iprobe	103.041s	≈ 7.5%
mca_pml_ob1_recv_req_start	46.168s	≈ 3.4%
[Others]	821.171s	≈ 59.9%
Total	1371.171s	100%

TABLE 2.1: Hotspots in one `RingCyclotron` run.

functions accounts for around 20.8% of the time spent in this example. Apart from the other hotspots, which are library calls that we cannot easily improve, the remaining functions each account for less than 3.5% and are thus hardly worth considering on their own yet together they constitute almost 60% of the effort.

2.2.1.2 Top-Down

While hotspot analysis considers only the time spent executing the instructions inside the function itself, another approach is to consider ranking routines according to the total amount of CPU time spent inside the function and all of its subroutines. This top-down point of view becomes even more interesting when we consider that both of the highlighted hotspots are subroutines of the `ParallelCyclotronTracker::rk4` routine. According to a sample of 5 timings of the baseline code $34.7\% \pm 0.5\%$ of the execution time is spent in `ParallelCyclotronTracker::rk4` and its subroutines.

Of the remaining time another roughly 17% is spent in `ParallelCyclotronTracker::deleteParticle` and its subroutines where in this example almost all of the time is used to perform a minimum reduction over all processes. Another approximately 13% is spent in computing the self-fields using `PartBunch::computeSelfFields_cycl` wherein the most time is spent with scattering, gathering and calls to a `DFT` library. Since both of these functions account for less than half the time that `ParallelCyclotronTracker::rk4` accounts for and seem to consist mostly of parallelization overhead and library calls, they do not seem to make fitting optimization targets.

The remaining computation time is spent in functions that each account for less than 10% of the runtime and are thus hardly worth considering at all since the upper bound on the overall speedup one could get from optimizing such functions is less than 1.12×.

We therefore elect to attempt to improve performance in `ParallelCyclotronTracker::rk4` for a maximal theoretical speedup of $\frac{1}{1-0.347} \approx 1.53$ if only this function is improved. While this is not a very satisfactory bound, it is the best we can do for this specific problem arising from this extensive real-world application. Furthermore, we remain open to the idea of also applying optimizations to different parts of the code if the opportunity presents itself.

2.2.2 Problem Size

This section attempts to justify the selection of 10^6 particles for the `RingCyclotron` measurements, which form the basis for the analysis in Section 2.2.1 and the selection of `ParallelCyclotronTracker::rk4` as the function to be

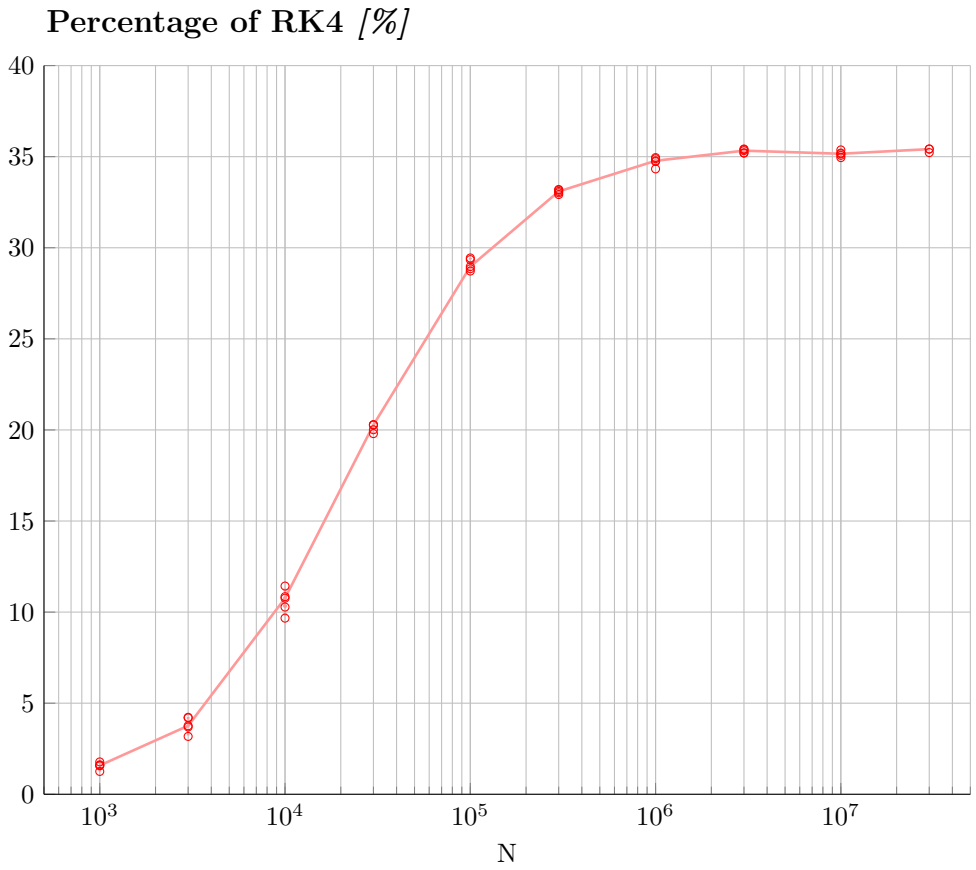

FIGURE 2.1: Percentage of CPU time spent in `ParallelCyclotronTracker::rk4` and its subroutines compared to the number of particles tracked.

optimized, by investigating the percentage of the total runtime taken up by `ParallelCyclotronTracker::rk4` with increasing number of particles.[8] For small numbers of particles the percentage of time spent executing the tracking, and therefore also the time-stepping, is expected to decrease as the overhead of the simulation begins to dominate. Conversely, with increasing problem size one part of the tracker might start to dominate over the others as its computational effort scales worse.

As we can observe from Figure 2.1 the percentage of `ParallelCyclotronTracker::rk4` seems to flatten out to around 35% with increasing number of particles in this example. It therefore seems likely that at this point the tracker is dominating the overhead. Since the number of calls to `ParallelCyclotronTracker::rk4` increases linearly with, and the runtime of such a call is independent of, the number of particles, the time spent in `ParallelCyclotronTracker::rk4` increases linearly. Similarly the other parts of the tracking algorithm also scale linearly with the number of particles which supports the conjecture that the percentage of `ParallelCyclotronTracker::rk4` flattens out at a certain problem size.

[8] Only the input parameter `NPART` is increased while all other parameters are left as presented in Appendix B.

```
for (int i = 0; i < N; ++i) {
    x = x + v[i];
}
```

Code 2.1: Computation of the sum of a floating-point array `v`.

```
for (int i = 0; i < N; i += 4) {
    x0 = x0 + v[i];
    x1 = x1 + v[i + 1];
    x2 = x2 + v[i + 2];
    x3 = x3 + v[i + 3];
}
x0 = x0 + x1;
x2 = x2 + x3;
x = x0 + x2;
```

Code 2.2: Unrolled computation of the sum of a floating-point array `v`.

2.2.3 Conclusion

On one hand, we can quite clearly observe that `ParallelCyclotronTracker::rk4` is the best bet for optimizing because of its ratio of lines of code to time spent executing. On the other hand, we also already have a rather discouraging upper bound on the possible speedup if we only improved this one function which raises concerns of how well the reduced readability of the code will pay off. Additionally, that upper bound does not seem to scale up with the problem size.

2.3 Optimizations and Rationales

The following section is a recapitulation of some of the optimization techniques discussed in Prof. Marcus Püschel's lecture "How To Write Fast Numerical Code" [4]. It reviews those techniques that we think applicable to the computation bound problem at hand and which we plan on using to enable compiler optimizations. The application of these optimizations will be referred to as rewriting the code in *'compiler-friendly style'*, since the goal mostly is to enable compiler optimizations.

2.3.1 Loop Unrolling

Unrolling loops using accumulators can help improve instruction level parallelism[9] [4]. Consider as an example Code 2.1, where the sequential dependence among the iterations of the loop requires the code to be scheduled such that the next addition is only issued once the previous one has been completed. Under the assumption that `N%4 == 0`,[10] unrolling this loop using accumulators yields Code 2.2 performing four independent operations in each iteration, which can be issued together hiding the latency[11] of the instruction.

[9] Lecture: "Optimization for Instruction-Level Parallelism".
[10] If `N%4 != 0` the remainder is easily dealt with sequentially after the main iteration. This is omitted in this example for simplicity.
[11] Number of cycles one needs to wait until the instruction is finished from fetching to register write back.

```
for (int i = 0; i < N; i+=4) {
    v[i] += w[i];
    v[i + 1] += w[i + 1];
    v[i + 2] += w[i + 2];
    v[i + 3] += w[i + 3];
}
```

CODE 2.3: Partially unrolled loop adding a floating-point array `w` to `v`.

```
for (int i = 0; i < N; i += 4) {
    // Load
    double v0 = v[i], v1 = v[i + 1], v2 = v[i + 2], v3 = v[i + 3];

    // Compute
    v0 += w[i];
    v1 += w[i + 1];
    v2 += w[i + 2];
    v3 += w[i + 3];

    // Store
    v[i] = v0;
    v[i + 1] = v1;
    v[i + 2] = v2;
    v[i + 3] = v3;
}
```

CODE 2.4: Partially unrolled loop adding a floating-point array `w` to `v` using scalar replacement.

To determine by how much each loop should be unrolled one can consider the latency L and throughput[12] T. In the case of floating-point arithmetic and without unrolling or accumulators, the compiler must assume each iteration to depend on the previous one meaning that one has to wait a full L cycles before the instruction for the next iteration can be issued. Conversely, if all instructions are independent, the limiting factor becomes the throughput of the execution unit. This leads to the conclusion that as long as the number of accumulators K satisfies $K \geq LT$, pipelining is not hindered by the dependencies between the iterations. Since more accumulators only lead to code bloat, $K = \lceil LT \rceil$ would be optimal[13] [4].

2.3.2 Scalar Replacement

In this context, aliasing describes the problem that two pointers might point to the same memory location. Since this is not easy for a compiler to predict the compiler must expect aliasing all over the place [4], as is the case in Code 2.3, where the operations within one iteration might look independent at a first glance but one cannot expect the compiler to treat the iterations as such since the arrays might overlap in such a way that assuming independent operations would lead to incorrect results – e.g. `w[i + 1]` might reference the same memory location as `v[i]`. Conversely, if we follow the explanations in [4] to dereference the pointers into scalars with which we do the computation and store these scalars at the pointed-to locations afterwards, we remove this possibility for aliasing (see Code 2.4) enabling those values to be allocated to registers and allowing the compiler to use instruction scheduling. Of course, this substitution only produces correct results if in fact there was no aliasing in the first place, meaning that by using this replacement, the programmer simply writes the code in a style that allows the compiler not to conservatively expect aliasing.

[12] Number of executions that can be done per cycle.
[13] See Footnote 9, Slide 22.

```
// Read after write
c = a + b;
d = e + c;

// Write after write
c = a + b;
foo(c);
c = e + d;

// Write after read
c = a + b;
a = e + d;
```

CODE 2.5: Examples of dependencies.

2.3.3 Single Assignment

In the snippets in Code 2.5, one can see that in each of the examples there exists a dependency of the last line on the first in some way as indicated by the preceding comment. However, the only necessary dependency is the first one with all others being only dependencies by name. As presented in [4], using a different variable name for the last assignment removes such a dependency allowing the compiler to make use of more optimizations. Dependencies by name can be completely avoided by adhering to the single assignment rule enforcing that each variable is only assigned once, which can easily be checked by declaring it `const` in `c++`. Since a `const` variable will never be the left-hand side of an assignment more than once it can also never be the source of write-after-write or write-after-read dependencies. Unfortunately, this way one tends to run out of meaningful, and by extension self-documenting, variable names very quickly.

2.3.4 Code Motion, Common Subexpressions Elimination & Strength Reductions

We also introduce optimizations that the compiler is likely to apply automatically in order to reduce the number and strength of the computations as outlined[14] in [4]. Since we also wish to introduce vectorization with `intrinsics` we fear that the compiler might not realize these optimizations in that context by itself.

2.3.5 Inlining

Finally, we inline all static function calls when applicable. This is done not primarily because we expect inlined code to be faster by saving on the function call – we are very well aware of the fact that inlining might slow down execution not least because the instruction cache will need to hold more instructions. The inlining is done in the hopes that it might enable further optimizations by the compiler if the calling context is known at compile time. As with the previous optimizations the additional information might allow the compiler to refrain from being overly conservative.

[14] Lecture: "Benchmarking, Compiler Limitations".

```
#include<tuple>
#include<string>
using RGB = std::tuple<double, double, double>;

enum Color {RED, GREEN, BLUE = 0};

std::string toString(Color c) {
    switch (c) {
         case RED: return "RED";
         case GREEN: return "GREEN";
         case BLUE: return "BLUE";
         default: throw std::invalid_argument("Unknown color");
    }
}

RGB toRGB(Color c) {
    switch (c) {
        case RED: return RGB(1,0,0);
        case GREEN: return RGB(0,1,0);
        case BLUE: return RGB(0,0,1);
        default: throw std::invalid_argument("Unknown color");
    }
}
```

Code 2.6: Definition of an `enum` and functions referencing each of its `enumerator`s.

```
// Helper macros to process single enumerator
#define ENUM_VALUE(name, assign, ...)  name assign,
#define ENUM_STRING(name, ...) case name: return #name;
#define ENUM_RGB(name, assign, r,g,b) case name: return RGB(r,g,b);

// macros to define functionality
#define DEFINE_ENUM(EnumName,CB) \
    enum EnumName { \
        CB(ENUM_VALUE) \
    };

#define DEFINE_TOSTRING(EnumName, CB) \
    std::string toString (EnumName c) { \
         switch (c) { \
             CB(ENUM_STRING) \
             default: throw std::invalid_argument("Unknown color"); \
         } \
    };

#define DEFINE_TORGB(EnumName, CB) \
    RGB toRGB (EnumName c) { \
         switch (c) { \
             CB(ENUM_RGB) \
             default: throw std::invalid_argument("Unknown color"); \
         } \
    };
```

Code 2.7: Definition of the factory `macros` used to create Code 2.6.

```
// Definition of enums
#define DEFINE_COLOR(CB) CB(RED, ,1,0,0) CB(GREEN, ,0,1,0) CB(BLUE,=0,0,0,1)
#define DEFINE_DAY(CB) CB(Monday,) CB(Tuesday,) CB(Wednesday,) CB(Thursday,) CB(Friday,) CB(Saturday,) CB(Sunday,)

// Invocations of the factory
DEFINE_ENUM(Color,DEFINE_COLOR)
DEFINE_TOSTRING(Color,DEFINE_COLOR)
DEFINE_TORGB(Color,DEFINE_COLOR)

DEFINE_ENUM(Day, DEFINE_DAY)
DEFINE_TOSTRING(Day, DEFINE_DAY)
```

Code 2.8: Invocation of the factory to create Code 2.6.

```
std::string toString (Color c) { \
        switch (c) { \
            DEFINE_COLOR(ENUM_STRING) \
            default: throw std::invalid_argument("Unknown color"); \
        } \
    };
```

Code 2.9: First step in the expansion of `DEFINE_TOSTRING(Color, DEFINE_COLOR)`.

```
std::string toString (Color c) { \
        switch (c) { \
            ENUM_STRING(RED, ,1,0,0) ENUM_STRING(GREEN, ,0,1,0) ENUM_STRING(BLUE,=0,0,0,1) \
            default: throw std::invalid_argument("Unknown color"); \
        } \
    };
```

Code 2.10: Second step in the expansion of `DEFINE_TOSTRING(Color, DEFINE_COLOR)`.

2.4 Enum Factory

2.4.1 Problem

Consider a scenario where one wishes to define an `enum E` as well as some functionality that references each `enumerator` in `E`, like for example in Code 2.6. This snipped seems a bit error prone as each `enumerator` (such as `GREEN`) has to be typed out multiple times and thus there is a possibility of breaking the code when extending and maintaining by mistyping or forgetting one of the occurrences. Adding more functions which reference each `enumerator` in a similar manner would make this issue even more prevalent. To reduce the fragility of this code by allowing it to be created without typing the `enumerators` multiple times one can make use of a collection of `macros` which serve as a sort of `enum` factory.

2.4.2 Factory

The following factory is constructed in the same way as [5] constructs their solution to a similar but slightly less general problem and the resulting code follows directly from the ideas presented in [5]. In this discussion function-like `macros` are treated as if they were functions which is a helpful view when explaining the factory. The factory consists of several `macros` which take other `macros` as parameters resembling a callback pattern. For each functionality a `macro` is defined which generates the general structure of the function or definition and then delegates the `enumerator` mentions to a callback `macro` to be passed as a parameter. This `macro` parameter must itself be passed a `macro` as argument whereto it will apply every `enumerator`. As an example we present in Code 2.7 the `macros` needed to create the snippet in Code 2.6.

2.4.3 Invocation

To use the factory the user must define their own `macro` which again takes a callback and invokes it with each `enumerator` and invoke the factory `macros` as presented in Code 2.8.

2.4.4 Mechanism

To understand the mechanism we trace the expansion of `DEFINE_TOSTRING(Color, DEFINE_COLOR)` from Code 2.8. The `macro` would first expand to Code 2.9 according to the definition in Code 2.7 and would then substitute `DEFINE_COLOR(ENUM_STRING)` according to the `macro` in Code 2.8 to arrive at Code 2.10. After expanding the `ENUM_STRING` occurrences we would end up with the desired code. The same approach can be used to trace the expansions of the other factory invocations.

Chapter 3
Optimizations

In this chapter, we outline the rather aggressive optimizations we apply to the `RingCyclotron` example and especially to the subroutines of `ParallelCyclotronTracker::rk4`, through rewriting parts of the `OPAL` code in *'compiler-friendly style'* described in Chapter 2. To be able to track the improvement in computation time and deterioration in readability this is done in six major stages: application of *'compiler-friendly style'* to, and inlining of, `ParallelCyclotronTracker::derivate` and `Cyclotron::apply`, reduction of string comparisons and finally vectorization.

3.1 Baseline Implementation

For the baseline implementation[15] all that is changed compared to the original implementation is the removal of a timing instrumentalization.[16] It would not be entirely fair to consider turning such an instrumentalization off as an optimization, therefore comparing to this baseline implementation seems fairer. It should also be noted that the optimization flag `-march=native`[17] is added to the compilation command for the baseline code. Since this flag is used for all the optimized implementation it seems unfair to have it lacking from the baseline implementation as one would be claiming responsibility for a speedup which comes solely from the compiler rather than the programmer.

```
yp[3] = (externalE(0) / c  + (externalB(2) * y[4] - externalB(1) * y[5]) / tempgamma) * qtom; // [1/ns]
```

Code 3.1: Original line 4040 in `ParallelCyclotronTracker.cpp`.

```
const double y3 = y[3];
// ...
// yp[3] = (externalE(0) / c  + (externalB(2) * y[4] - externalB(1) * y[5]) / tempgamma) * qtom; // [1/ns]
const double t011 = 1.0 / c;
const double t012 = b2 * y4;
const double t013 = b1 * y5;
const double t014 = e0 * t011;
const double t015 = t012 - t013;
const double t016 = t015 * t007;
const double t017 = t014 + t016;
const double yp3 = t017 * qtom;
// ...
yp[3] = yp3;
```

Code 3.2: Code 3.1 after applying *'compiler-friendly style'*.

[15] In this context a baseline implementation is a correctly working version of the code which serves as a fair point of comparison to assess how much improvement the optimizations yield.
[16] The code for this version can be found in the `opt0` branch.
[17] See Footnote 7.

```
// Line 1093
*gmsg << "* -> " <<  ((*sindex)->first) << endl;
// Line 2440
if(((*sindex)->first) == "CAVITY") {
```

CODE 3.3: Lines 1093 and 2440 of `ParallelCyclotronTracker.cpp` of the baseline implementation.

```
enum ComponentType { CYCLOTRON , RING , OPALRING , PROBE , COLLIMATOR , STRIPPER , SEPTUM , CAVITY , };
inline const char *getString(ComponentType value) {
    switch(value) {
        case CYCLOTRON: return "CYCLOTRON";
        case RING: return "RING";
        case OPALRING: return "OPALRING";
        case PROBE: return "PROBE";
        case COLLIMATOR: return "COLLIMATOR";
        case STRIPPER: return "STRIPPER";
        case SEPTUM: return "SEPTUM";
        case CAVITY: return "CAVITY";
        default: return "";
    }
}
```

CODE 3.4: An `enum` to represent component types and a function to map `enumerator`s to their name.

```
// Line 1093
*gmsg << "* -> " <<  getString((*sindex)->first) << endl;
// Line 2440
if(((*sindex)->first) == ComponentType::CAVITY) {
```

CODE 3.5: Using the `enum` in Code 3.4 for the use cases in Code 3.3.

3.2 'Compiler-Friendly Style' in `ParallelCyclotronTracker::derivate`

The first step towards optimized and vectorized code is taken by applying the single static assignment rule to the code in `ParallelCyclotronTracker::derivate` and performing some code motion to obtain a much uglier version of the code[18] which should be easier to optimize for the compiler.

As an example we present the change of line 4040 in `ParallelCyclotronTracker.cpp` in Codes 3.1 and 3.2. One can already see that this constitutes quite an obfuscation and was it not for the comment containing the original code it would be very hard to recognize the line or deduce its purpose. However, as we will establish later, this first rewriting provides a decent speedup for the overall computation time of the `RingCyclotron` example already. In this very first optimization step, we therefore already strike at the heart of the problem encountered when optimizing such an involved and lengthy code with aggressive application of *'compiler-friendly style'*, namely that there is a price to be payed in terms of readability in return for the speedup.

3.3 Reducing String Comparisons

In this step,[19] we do not work with the hotspots exclusively but rather we improve overall performance with a simple substitution of `std::string`s with `enum`s for a property which serves as a 'tag' indicating what type a certain

[18] See `opt1_ssa_derivate` branch.
[19] See `opt2_reduce_string_compares` branch.

```
beamline_list::iterator sindex = FieldDimensions.begin();
bool outOfBound = (((*sindex)->second).second)->apply(Pindex, t, externalE, externalB);
```

Code 3.6: Virtual function call to `apply` in `ParallelCyclotronTracker::derivate`.

```
// always put cyclotron as the first element in the list.
if(ElementType == "OPALRING" || ElementType == "CYCLOTRON") {
    sindex = FieldDimensions.begin();
} else {
    sindex = FieldDimensions.end();
}
FieldDimensions.insert(sindex, localpair);
```

Code 3.7: Code ensuring that call in Code 3.6 will only be applied to one of two objects.

component is. As an example of the use cases we wish to optimize consider Code 3.3. Looking at the rest of the code, one can easily deduct that the left-hand side of the comparison in line `2440` in Code 3.3 can only be assigned a value from a fairly limited set of values known at compile time. Therefore, this performs a string comparison for something that could more or less easily be substituted by an integer comparison. To remove these unneeded string comparisons we make use of the `enum` factory `macro` introduced in Section 2.4 to define the `enum` and functionality in Code 3.4 and change Code 3.3 to Code 3.5. The second line now compares `enum`s rather than `std::string`s whereas the printing now requires an additional (easily inlined) function call which consists only of a switch-case.

3.4 'Compiler-Friendly Style' in `Cyclotron::apply`

We apply the same rewriting techniques as in Section 3.2 to the code in `Cyclotron::apply` again trading performance for readability.[20]

3.5 Inlining `Cyclotron::apply`

In `ParallelCyclotronTracker::derivate` there is the virtual function call shown in Code 3.6. Generally speaking, virtual function calls are not possible to inline, since the actual function to be called might only be determinable at runtime. Recalling the top-down and hotspot analyses in Section 2.2.1, one might suspect that this `apply` call refers to the `Cyclotron::apply` function therein mentioned. In fact, tracing the way that the `FieldDimensions` container is filled in `ParallelCyclotronTracker::buildupFieldList` one observes the lines presented in Code 3.7, which allow one to be sure that – if `FieldDimensions` is filled using this member function and there is an element of the type `"OPALRING"` or `"CYCLOTRON"` added – the `apply` call will be performed on one of two objects. This would be enough indication to introduce a switch with the two special cases with static linking and a default case with dynamic linking. Fortunately, in the `RingCyclotron` example the used element is always of the type `"CYCLOTRON"` which confirms the initial hunch. Instead of changing the dynamic linking simply to static linking we inlined the function call entirely.[21]

[20] See `opt3_ssa_cyclotron_apply` branch.
[21] See `opt4_inline_cyclotron_apply` branch.

```
double memory[Dimension];
for (int i = 0 ; i < numberOfParticles; ++i) {
    prepare(i, memory);
    rk4(i,memory);
    bool condition = process(memory);
    if (condition) {
        rk4(i,memory);
    }
}
```

CODE 3.8: Very simplified structure of original code calling `ParallelCyclotronTracker::rk4`.

```
double memory[4 * Dimension];
int mask[4];
for (int i = 0 ; i < N; i+=4) {
    prepareParticles(i,memory);
    rk4_v(i,memory);
    process(memory, mask);
    if (any(mask)) {
        rk4_v(i,memory,mask);
    }
}
```

CODE 3.9: Reordering of Code 3.8.

3.6 Inlining `ParallelCyclotronTracker::derivate`

As a final change before applying vectorization we inline all four calls to `ParallelCyclotronTracker::derivate` in `ParallelCyclotronTracker::rk4` yielding one large piece of spaghetti code[22] for vectorization.

3.7 Vectorization

3.7.1 Floating-Point Precision

Since we do not wish to sacrifice the precision of the computation, and therefore choose to let all floating point numbers remain double precision, the parallelism that can be gained through vectorization is limited such that with `AVX` support only four particles can be handled at once.

3.7.2 Reordering

Vectorizing `ParallelCyclotronTracker::rk4` requires a reordering both inside and outside of that function. On a very simplified and abstracted level the structure in the original code looks something along the lines of Code 3.8[23] which is then reworked to Code 3.9. Furthermore, the code inside `ParallelCyclotronTracker::rk4` must be moved to `ParallelCyclotronTracker::rk4_v` and reworked such that it performs the computation for up to four particles at once using the vector instructions.

[22] See `opt5_inline_derivate` branch.

[23] Note that in the `RingCyclotron` example the conditional part seems to be executed for less than 1% of the iterations, meaning that `condition` is usually `false`.

```
/*
 * Returns one sample of distribution f over [range[0], range[1]]
 * generated through rejection sampling with uniform proposal distribution.
 */
double rejection_sample(double(*f)(double), double range[2]) {
    for (int i = 0; i < MAX_REJECTIONS; ++i) {
        double proposal = range[0] + (range[1] - range[0]) * get_uniform_sample();
        if (f(proposal) > get_uniform_sample()) {
            return value;
        }
    }
    // Error handling
}
```

Code 3.10: Scalar implementation of rejection-sampling.

3.7.3 Guided Auto-Vectorization

`OpenMP4.0` supports guided auto-vectorization using `#pragma omp simd` statements [6]. We attempt to make use of these in three ways:

1. Simply guide the compiler towards vectorizing the outermost loop in the code without the reordering of Section 3.7.2 while declaring a vectorized version of `ParallelCyclotronTracker::rk4`.
2. Apply the reordering as in Section 3.7.2, then guide the compiler towards vectorizing a loop over multiple calls to `ParallelCyclotronTracker::rk4`.
3. Exchange that loop with a single call to `ParallelCyclotronTracker::rk4_v`, which is implemented as a loop over four particles wrapping around the code of `ParallelCyclotronTracker::rk4`. Guide the compiler towards vectorizing that inner loop.

None of this yields the desired vectorization presumably because the code to be vectorized is much more involved – several hundred lines of code with branching – than the simple `#pragma` statement can handle. It should be noted, however, that the compiler has been applying some vectorization measures itself as discussed in Section 4.3.4.

3.7.4 Vectorization Using Compiler Intrinsics

`ParallelCyclotronTracker::rk4` is reworked into `ParallelCyclotronTracker::rk4_v`, such that it can process four particles at once rather than only one and fits into the reordered calling context in Section 3.7.2, using compiler `intrinsics` [7].[24]

3.7.4.1 SVML

We use `Intel`'s short vector math library (`SVML`), which is included with `icc` [8], to compute trigonometric functions for four double precision floating point numbers at once.

[24] See `vec` branch.

```
__m256d rejection_samples(__m256d (*f)(__m256d), __m256d range[2]) {
    __m256d mask = FALSE;
    __m256d proposal;
    for (int i = 0; i < MAX_REJECTIONS; ++i) {
        proposal = range[0] + (range[1] - range[0]) * get_uniform_samples();
        mask = f(proposal) > get_uniform_samples();
        if (all(mask)) return value;
        if (any(!mask)) {
            for (int j = 0; j < 4; ++j) {
                if (mask[j]) {
                    proposal[j] = range[0][j] + (range[1][j] - range[0][j]) * get_uniform_sample();
                }
            }
        }
    }
    // Error handling
}
```

Code 3.11: Partially vectorized rejection-sampling, which handles conditional part serially.[25,26]

```
__m256d rejection_samples(__m256d (*f)(__m256d), __m256d range[2]) {
    __m256d mask = FALSE;
    __m256d proposal = _mm256_set1_pd(0.0);
    for (int i = 0; i < MAX_REJECTIONS; ++i) {
        proposal = _mm256_blendv_pd(range[0] + (range[1] - range[0]) * get_uniform_samples(),proposal,mask);
        mask = _mm256_or_pd(mask,f(proposal) > get_uniform_samples());
        if (all(mask)) return value;
    }
    // Error handling
}
```

Code 3.12: Fully vectorized version of Code 3.10 using masking.[27]

3.7.4.2 Masking

When trying to vectorize branching code we run into problems when a branch must be followed by only some of the elements of the vector as for example in Code 3.10. A very naïve way to handle this branching is to simply process the branches serially as is presented in Code 3.11. This approach might be justified if the conditional code is cheap to compute, the condition is only rarely satisfied for more than one element in the vector and the conditional part features itself a lot of branching statements – such that by handling it serially, these inner branches do not need to be handled in any special way. Otherwise, however, one forgoes a large part of the vectorization speedup, as is the case if we handle the Code 3.10 in such a way. Additionally, the overhead of extracting single entries out of all vector registers involved to be able to start the serial computation might also take a significant amount of time.

As an alternative consider computing the conditional part for all elements of the vector even if the condition is not satisfied – possibly taking care of invalid inputs to the conditional part by setting them to some dummy value – and blending the result according to the mask as in Code 3.12. This way we do not need to extract elements from the vectors and can do all operations vectorized. The drawback is of course that it performs useless computation for those particles that do not meet the condition potentially negating any speedup if it just so happens that in every group

[25] Note that the `gcc` extension allowing to use the subscript operator to access a single element of a vector is not supported in `icc` where this code would not compile. Since there is a wordy workaround for this issue we choose to ignore it for simplicity in this and the following examples.

[26] This function does not compute the same values as multiple calls to the function in Code 3.10 would since the pseudo-random number generator is accessed in a different pattern. The results merely model samples of the same distribution.

[27] As in Footnote 26, note that this implementation again accesses the pseudo-random number generator differently thus yielding different results from the other implementations. In addition, this implementation 'wastes' random numbers by disregarding new random samples for elements that already satisfy the acceptance criterion.

```
/*
 * Returns the bilinear interpolation at point x of the function defined by
 * the samples.
 */
double bilinearInterpolation(double x, double[N*M] samples) {
    // Compute index
    const size_t i = std::floor(x*N) + N * std::floor(y*M);
    const size_t j = i + 1;
    const size_t k = i + N;
    const size_t l = l + 1;

    // Compute weight
    const double w = N*x - std::floor(N*x);
    const double v = M*y - std::floor(M*y);

    // Get sample values
    const double a = samples[i];
    const double b = samples[j];
    const double c = samples[k];
    const double d = samples[l];

    // Compute Interpolation
    return (1-w) * a + w * b + (1 - v) * c + v * d;
}
```

Code 3.13: Bilinear interpolation on $[0, 1)^2$.

```
__m256d bilinearInterpolation(__m256d x, double[N*M] samples) {
    // Compute index
    const __m128i i = _mm256_cvtpd_epi32(x*N) + N * _mm256_cvtpd_epi32(y*M);
    const __m128i j = i + 1;
    const __m128i k = i + N;
    const __m128i l = k + 1;

    // Compute weight
    const __m256d w = N*x - _mm256_floor_pd(N*x);
    const __m256d v = M*y - _mm256_floor_pd(M*y);

    // Get sample values.
    const __m256d a = samples[i];
    const __m256d b = samples[j];
    const __m256d c = samples[k];
    const __m256d d = samples[l];

    // Compute Interpolation
    return (1-w) * a + w * b + (1 - v) * c + v * d;
}
```

Code 3.14: An attempt to vectorize Code 3.13.

```
// Get sample values.
const __m256d a = _mm256_set_pd(samples[i[3]],samples[i[2]],samples[i[1]],samples[i[0]]);
const __m256d b = _mm256_set_pd(samples[j[3]],samples[j[2]],samples[j[1]],samples[j[0]]);
const __m256d c = _mm256_set_pd(samples[k[3]],samples[k[2]],samples[k[1]],samples[k[0]]);
const __m256d d = _mm256_set_pd(samples[l[3]],samples[l[2]],samples[l[1]],samples[l[0]]);
```

Code 3.15: Cumbersome and slow workaround for gathering.

of four there is exactly one element that satisfies the condition. However, while in this worst case scenario masking degenerates to the prior case it is still advantageous in the other cases making it the better choice.

3.7.4.3 Shuffling

Gathering

To interpolate a set of samples at an arbitrary point, some values will need to be loaded from memory. The location of

```
dst[127:0] := MEM[loaddr+127:loaddr]
dst[255:128] := MEM[hiaddr+127:hiaddr]
dst[MAX:256] := 0
```

Code 3.16: Documentation of _mm256_loadu2_m128d from [7].

```
// Get sample values.
const __m256d a_prime = _mm256_loadu2_m128d(&samples[i[0]],&samples[k[0]]);
const __m256d b_prime = _mm256_loadu2_m128d(&samples[i[1]],&samples[k[1]]);
const __m256d c_prime = _mm256_loadu2_m128d(&samples[i[2]],&samples[k[2]]);
const __m256d d_prime = _mm256_loadu2_m128d(&samples[i[3]],&samples[k[3]]);

// Transpose such that a == _mm256_set_pd(a_prime[3],b_prime[2],c_prime[1],d_prime[0]) etc.
```

Code 3.17: Workaround for gathering using `loadu_w` and a matrix transpose.

```
/*
 * Interprets the entries of the array A as rows of a matrix and stores
 * elements representing the rows of the transposed matrix into B.
 */
void _mm256_transpose_pd(const __m256d A[], __m256d B[]) {
    // [A[0][0] A[1][0] A[0][2] A[1][2]]
    // [A[0][1] A[1][1] A[0][3] A[1][3]]
    // [A[2][0] A[3][0] A[2][2] A[3][2]]
    // [A[2][1] A[3][1] A[2][3] A[3][3]]
    __m256d lo_shuffle_tmp_0_1 = _mm256_unpacklo_pd(A[0],A[1]);
    __m256d hi_shuffle_tmp_0_1 = _mm256_unpackhi_pd(A[0],A[1]);
    __m256d lo_shuffle_tmp_2_3 = _mm256_unpacklo_pd(A[2],A[3]);
    __m256d hi_shuffle_tmp_2_3 = _mm256_unpackhi_pd(A[2],A[3]);

    // [A[0][0] A[1][0] A[2][0] A[3][0]]
    // [A[0][1] A[1][1] A[2][1] A[3][1]]
    // [A[0][2] A[1][2] A[2][2] A[3][2]]
    // [A[0][3] A[1][3] A[2][3] A[3][3]]
    B[0] = _mm256_permute2f128_pd(lo_shuffle_tmp_0_1,lo_shuffle_tmp_2_3,0x20);
    B[1] = _mm256_permute2f128_pd(hi_shuffle_tmp_0_1,hi_shuffle_tmp_2_3,0x20);
    B[2] = _mm256_permute2f128_pd(lo_shuffle_tmp_0_1,lo_shuffle_tmp_2_3,0x31);
    B[3] = _mm256_permute2f128_pd(hi_shuffle_tmp_0_1,hi_shuffle_tmp_2_3,0x31);
}
```

Code 3.18: Using `intrinsics` to implement a short matrix transpose.

these values is conditional on the interpolation point. For an example consider Code 3.13 and an attempt to vectorize it to run for multiple interpolation points at once in Code 3.14.

The issue is that Code 3.15 will not compile since the subscript operator for double arrays is not overloaded for the argument type `__m128i` and we will need to implement this gathering ourselves.[28]

`AVX2` offers such an instruction in `vgatherdpd` accessible through the `_mm_i32gather_pd intrinsic` [7]. Unfortunately, if only `AVX` is available[29] one would have to resort to extracting the individual indices from the vector of indices, fetch the required values from memory and then load them into an array like in Code 3.15.

Luckily for the bilinear interpolation in Code 3.10, we see that `j == i + 1 && l == k + 1`. Using the `AVX intrinsic __m256d _mm256_loadu2_m128d(double const* hiaddr, double const* loaddr)`,[30] which allows loading from two

[28] Furthermore, there are expressions that mix scalars and vectors and the arithmetic operators are not overloaded for type `__m128i`. These issues can easily be fixed by more wordy code that does not contribute to the understanding of the problem to be discussed and are therefore ignored for simplicity.

[29] This is the case on the machine used for the profiling in Sections 2.2.1 and the measurements in Section 2.2.2 and Chapters 4 and 5. See Footnote 4.

[30] See the documentation from [7] in Code 3.16.

```
double distance(double p[N][3], int i) {
    double x = p[i][0];
    double y = p[i][1];
    double z = p[i][2];
    return std::sqrt(x*x + y*y + z*z);
}
```

CODE 3.19: Function which computes the distance from the origin for a given particle index.

```
double distance(double p[N][3], int i) {
    __m256d x = _mm256_set_pd(p[i + 3][0], p[i + 2][0], p[i + 1][0], p[i][0]);
    __m256d y = _mm256_set_pd(p[i + 3][1], p[i + 2][1], p[i + 1][1], p[i][1]);
    __m256d z = _mm256_set_pd(p[i + 3][2], p[i + 2][2], p[i + 1][2], p[i][2]);
    return _mm256_sqrt_pd(x*x + y*y + z*z);
}
```

CODE 3.20: Vectorization of Code 3.19.

```
double distance(double p[N * 3], int i) {
    __m256d x_prime = _mm256_loadu_pd(&p[i][0]);
    __m256d y_prime = _mm256_loadu_pd(&p[i + 1][0]);
    __m256d z_prime = _mm256_loadu_pd(&p[(i + 2)][0]);

    // Transpose such that x == _mm256_set_pd(x_prime[3],x_prime[2],x_prime[1],x_prime[0]) etc.

    return _mm256_sqrt_pd(x*x + y*y + z*z);
}
```

CODE 3.21: Improved version of Code 3.20 avoiding accessing the vector element-wise.

memory addresses, we can implement the accessing of the sample values with four such loads and a matrix transpose,[31] which is hopefully a bit less costly. This very same problem is encountered and solved in the vectorization of the code of `Cyclotron::apply`.

Reordering Coordinates

Consider an array `p` which represents the spatial coordinates of some particles, such that the coordinates of particle `i` are at `p[i][0]`, `p[i][1]` and `p[i][2]`, and attempt to vectorize Code 3.19 such that it can handle four consecutive particles (see Code 3.20). As with the previous shuffling one can also avoid accessing the vector element-wise by loading and transposing as in Code 3.21.

A problem very similar to this is encountered and solved in line `4794` of `ParallelCyclotronTracker.cpp` on the `vec` branch where samples of the `E` and `B` fields need to be transformed in the same way.

[31] The matrix transposition can easily be done using `intrinsic` 'swizzles' as in Code 3.18.

Chapter 4
Speedup

In the following sections, we compare the percentage of time spent inside `ParallelCyclotronTracker::rk4`, the speedups and the CPU time of the program at the different stages both for `icc` and `gcc` wherever applicable. We also compare the effect of disabling compiler auto-vectorization via the `-fno-tree-vectorize` flag.

4.1 Measurements

4.1.1 Method

In the following figures, each position on the X-axis corresponds to one code version - accessible as a branch of the repository. The leftmost version represents the baseline code named `opt0`. From left to right the code features more and more optimization steps - hence the branch names `opt1` to `opt5` - culminating in the vectorization - with the branch containing the vectorized code called `vec`. The different colors and line styles correspond to different compilers and compiler settings. Different program versions are obtained by compiling different branches of the repository with

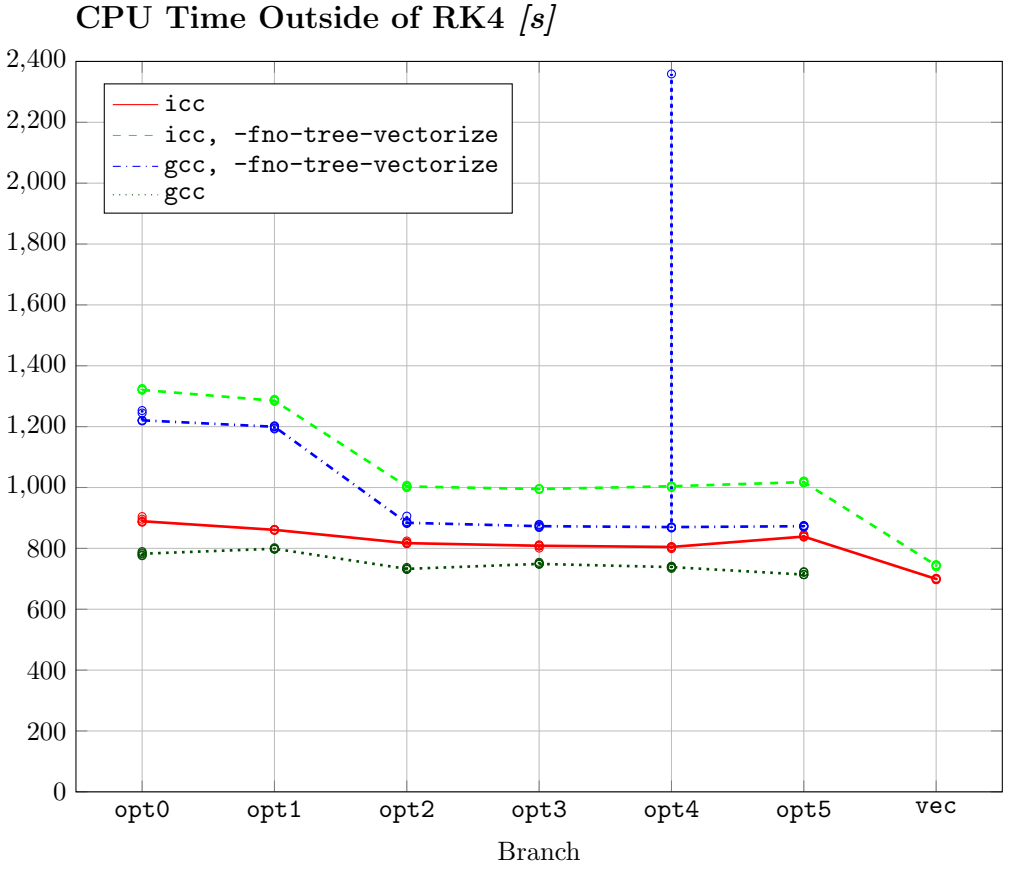

FIGURE 4.1: CPU time of functions other than `ParallelCyclotronTracker::rk4` and its subroutines.

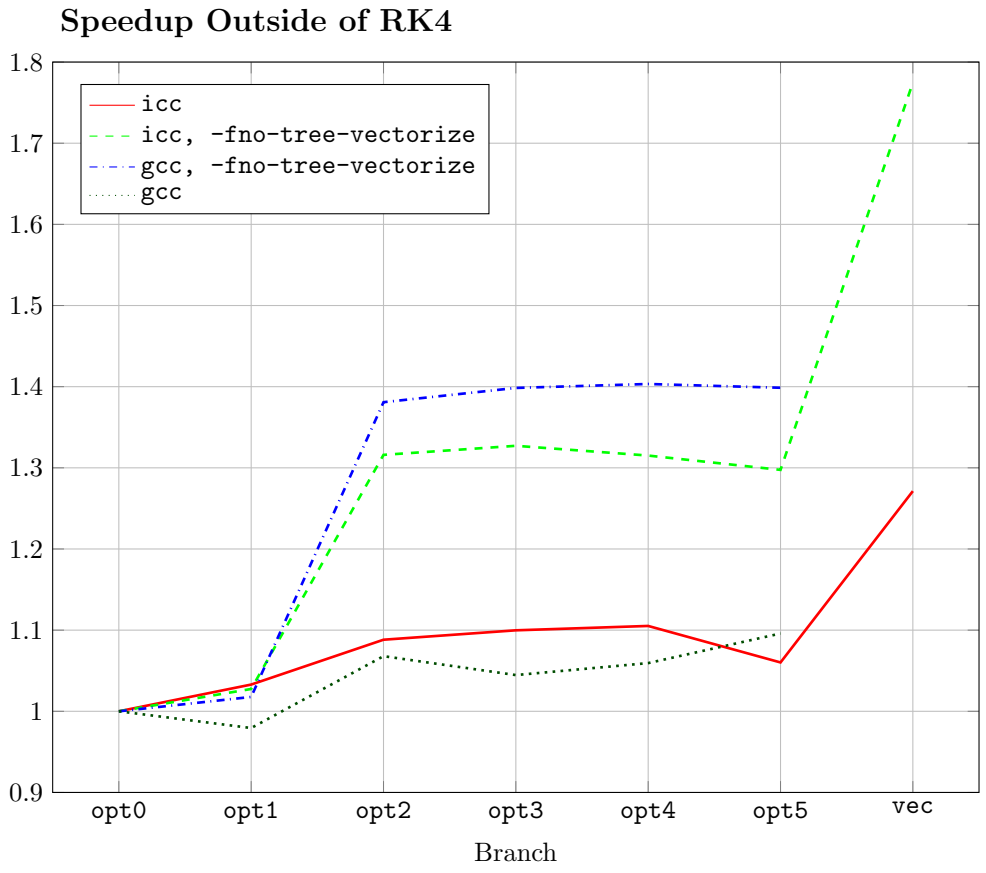

FIGURE 4.2: Speedup of functions other than `ParallelCyclotronTracker::rk4` and its subroutines over the baseline implementation.

different compilers and compiler settings. All measurements are collected running the `RingCyclotron` example[32] with 10^6 particles on the same 8 CPU machine.[33]

In Figures 4.1, 4.3, 4.5 & 4.6, we collect for each program version five measurements and mark their values as, sometimes overlapping, circles. Vertical lines, where visible, connect these measurements to highlight the range of values obtained by running the same code multiple times. Furthermore, for each program version the median of the five samples is computed and these medians are connected across branches.

For each compiler and settings, the speedup plots (Figures 4.2, 4.4 & 4.7) present the ratio of the medians of the CPU times of the baseline implementation `opt0` to the median of the CPU times of the program version in question.

4.1.2 Quality of Measurement

Note that the large outlier for `opt4` compiled with `gcc` and the `-fno-tree-vectorize` flag set corresponds to a measurement with significantly worse average CPU usage[34] of around 7.0 compared to a range in $[7.6, 7.9]$ for the

[32] See Footnote 3.
[33] See Footnote 4.
[34] Average CPU usage in this content is used to denote the average number of CPU threads performing work on the problem at any given time. On the machine used a maximum of 8.0 can be achieved with perfect concurrency (see Footnote 4).

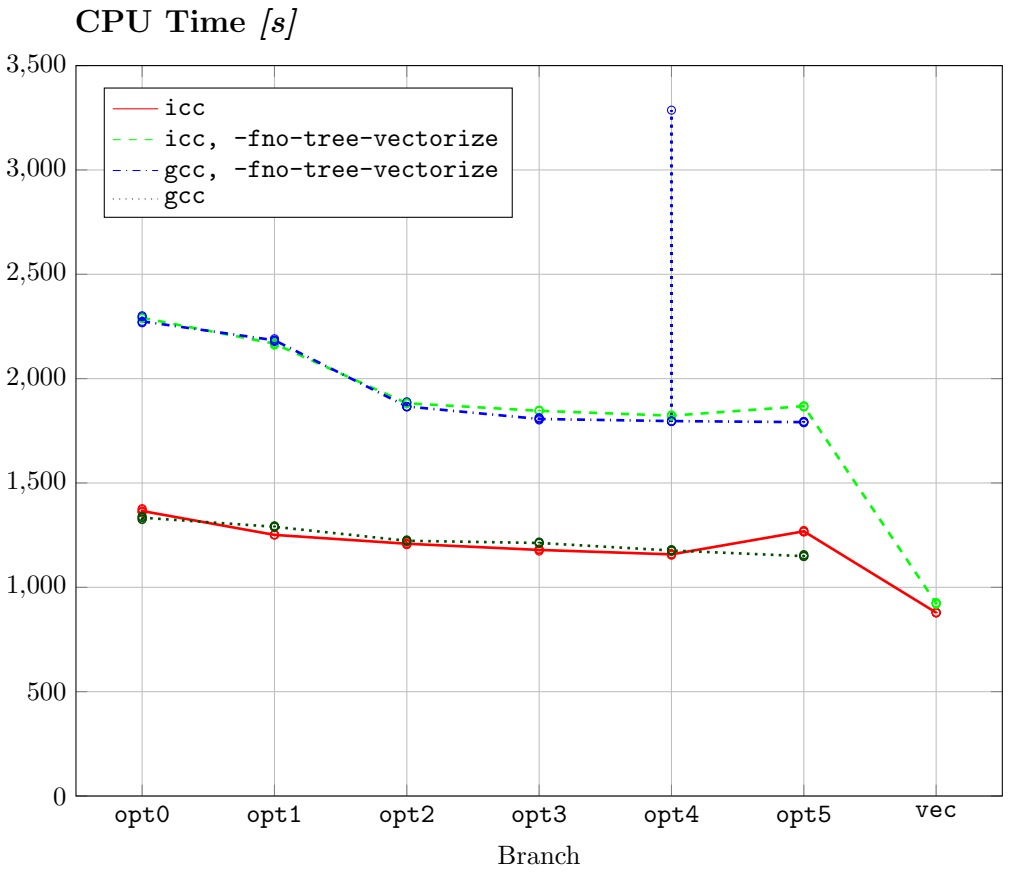

FIGURE 4.3: CPU time of the entire application.

other samples. We hypothesize that this is most likely due to load on the machine and thus an indication that this sample does not represent the range of the runtimes very well.

Apart from this one outlier, the individual measurements of each program version are bunched quite nicely together compared to the change in median across program versions. In other words, the lines in the plots seem meaningful in the sense that the differences of measurements across different program versions are much larger than the differences between measurements of the same versions.

4.2 Unexplained Speedups

4.2.1 Observations

Figures 4.1 and 4.2 show the time spent, and the speedup thereof, outside of `ParallelCyclotronTracker::rk4` and its subroutines. An expected observation is the large speedup that the replacement of the string comparisons yields since that optimization is performed on a global level and is therefore also expected to pay off globally. Unexpectedly, other optimization steps that exclusively targeted `ParallelCyclotronTracker::rk4` and its subroutines seem to have affected the runtime of the unchanged functions as well. Bearing in mind that we observe the median of five closely

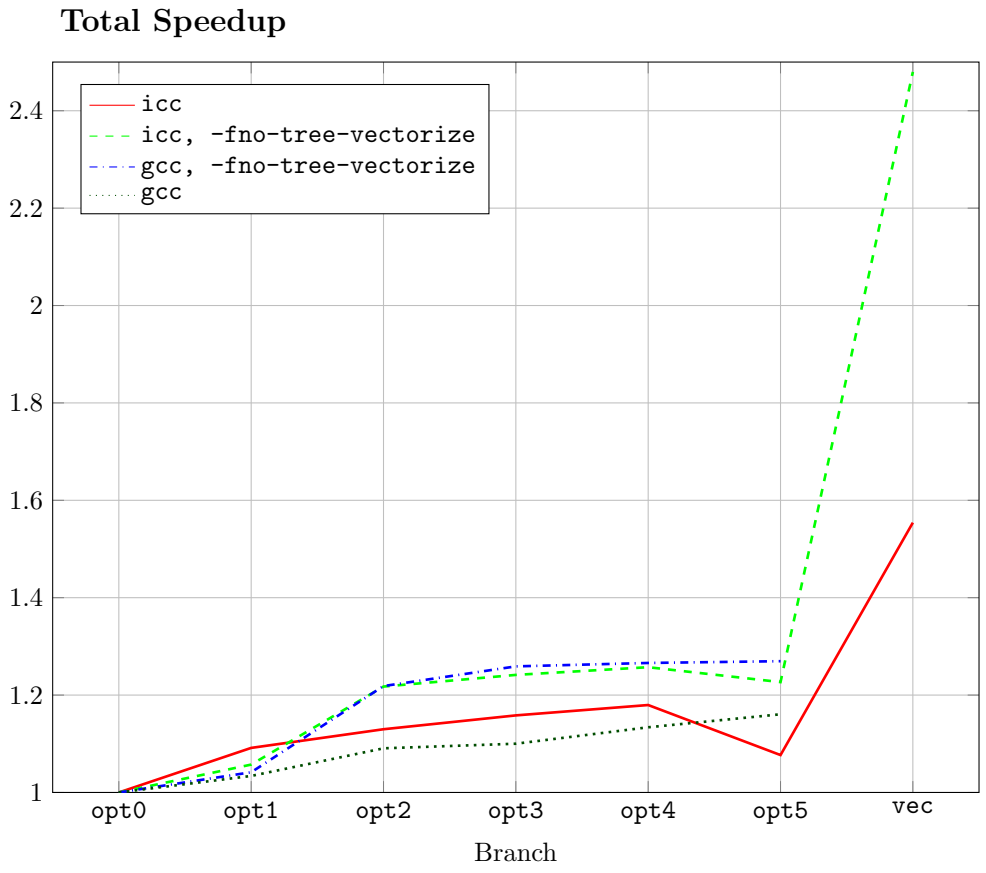

FIGURE 4.4: Speedup of the entire application over the baseline implementation.

bunched measurements – with the noted exception of one sample of an execution of `opt4` compiled with `gcc` and disabled auto-vectorization – for each program version, there seem to be unexplained speedups present.

4.2.2 Correctness

A first hunch would be that the unexplained speedups are due to the optimizations breaking the code in such a way that less computation has to be performed in the calling context of `ParallelCyclotronTracker::rk4` but a wrong result is yielded. To dismiss such concerns the entire regression test suite is run multiple times for each of the optimizations and compiler settings always yielding the expected results. This serves as a strong indication that at least the functionality tested by the regression test suite is still correct.

4.2.3 Investigation

Despite a small investigation no explanation could be found and verified as to why this effect is observed: Firstly, blaming inaccurate measurements, be it due to the stochastic nature of the profiler or interference of other processes, might explain the speedups observed when comparing branches `opt1`, `opt3`, `opt4` and `opt5`, but falls short of explaining why the observed effect for branch `vec` seems much larger than the variation across the samples. Secondly, comparing

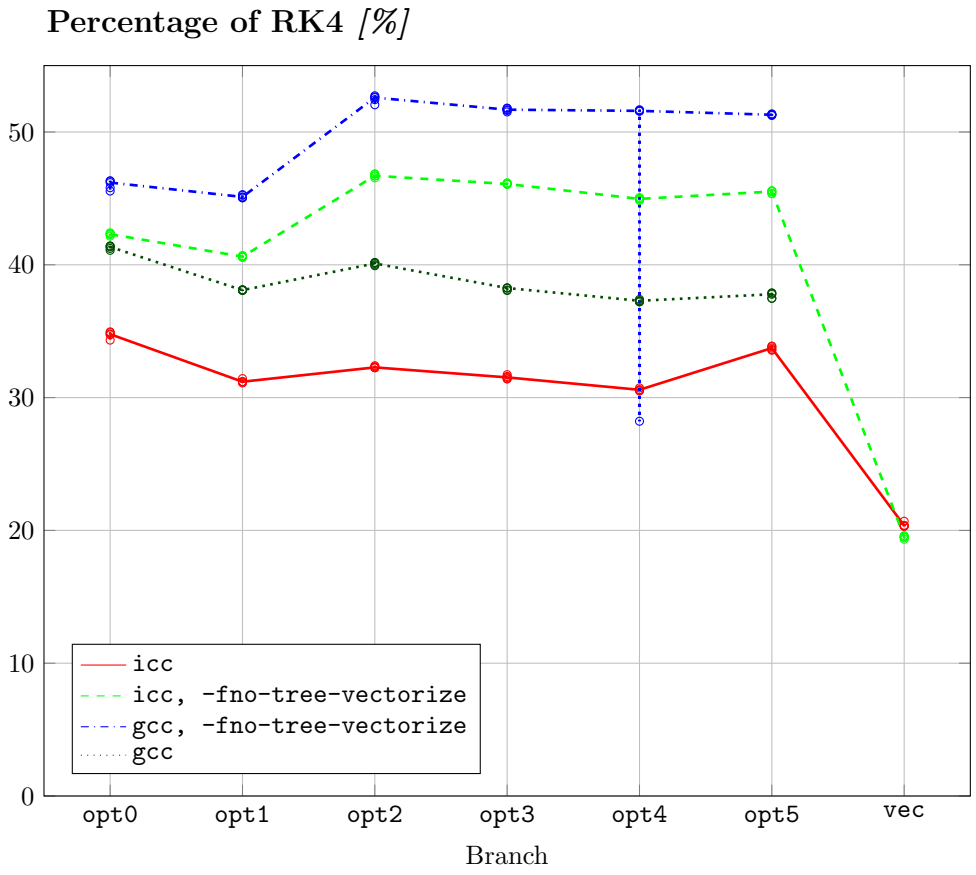

FIGURE 4.5: Percentage of CPU time spent in `ParallelCyclotronTracker::rk4` and its subroutines compared the rest of the application.

the executables created by the compilers across versions proves extremely difficult and unfruitful as the source code is not mapped to the executable in a way that allows to link the changes in source code to changes in the executable easily and sensibly due to the large amount of compiler optimizations enabled. Therefore, while comparing profiler results consistently points towards unexplained differences across different branches, any reasoning we could present as to what causes these speedups would be purely speculative.

4.2.4 Conclusion

We are fairly confident that `VTune Amplifier` consistently reports analyses pointing towards speedups outside of the function that is optimized but cannot explain the source of these speedups without resorting to speculation. Correctness of the computation nevertheless seems to be provided, according to the regression test suite. Therefore, we are fairly confident that the observed speedups are not due to mistakes that translate to faster yet incorrect code and may therefore be treated as a beneficial side-effect of whatever optimizations the compiler is able to apply.

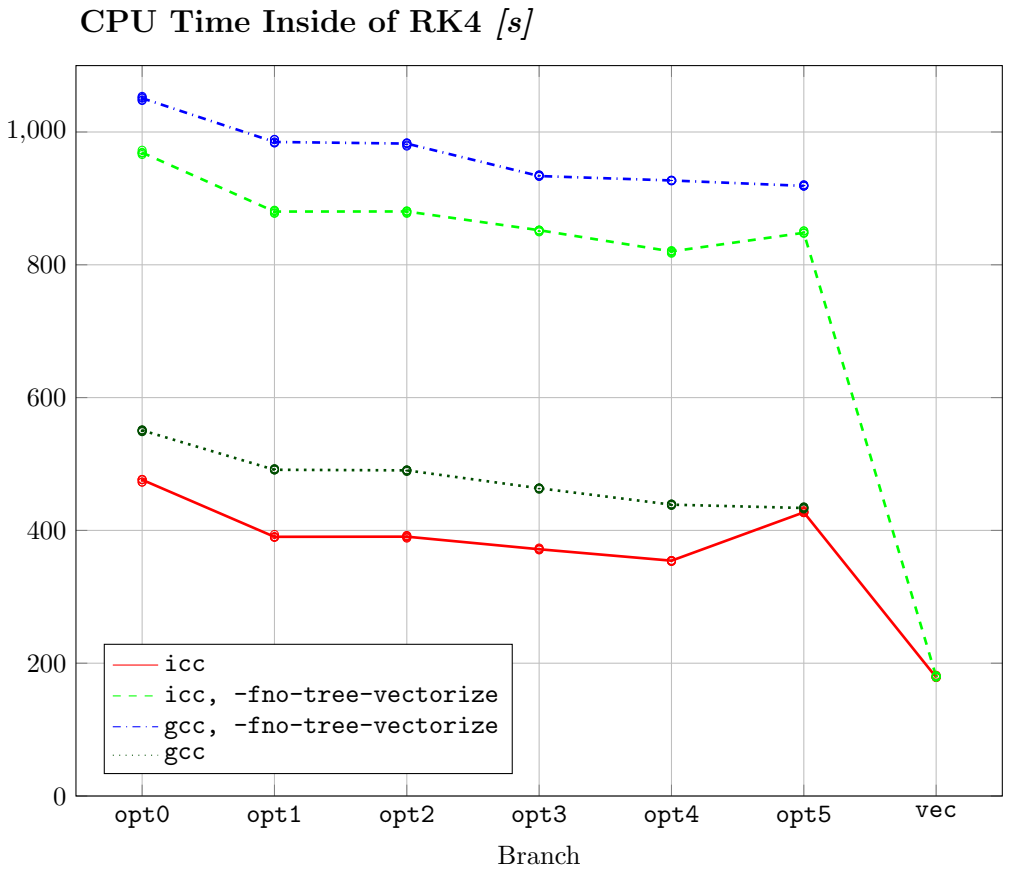

FIGURE 4.6: CPU time of `ParallelCyclotronTracker::rk4` and its subroutines.

4.3 Trends

4.3.1 Baseline Comparisons

We turn our attention towards the leftmost dots of Figures 4.3, 4.5 and 4.6 to discuss the runtimes of the baseline implementation `opt0` when compiled with different compilers and settings. As a first observation one can note that while differences caused by using different compilers are quite small, disabling the auto-vectorization has a very large impact on the overall performance of the baseline code. Figure 4.5 shows an interesting difference between the two compilers where it seems that the baseline code compiled with `gcc` spends significantly more of its time in `ParallelCyclotronTracker::rk4` compared to the version compiled with `icc`. Since the runtime differences are not as pronounced, we can conclude – and indeed observe in Figures 4.1 and 4.6 – that the code compiled with `icc` is quicker in `ParallelCyclotronTracker::rk4` and its subroutines than the code compiled with `gcc` while for the remaining code this situation is inverted on average.

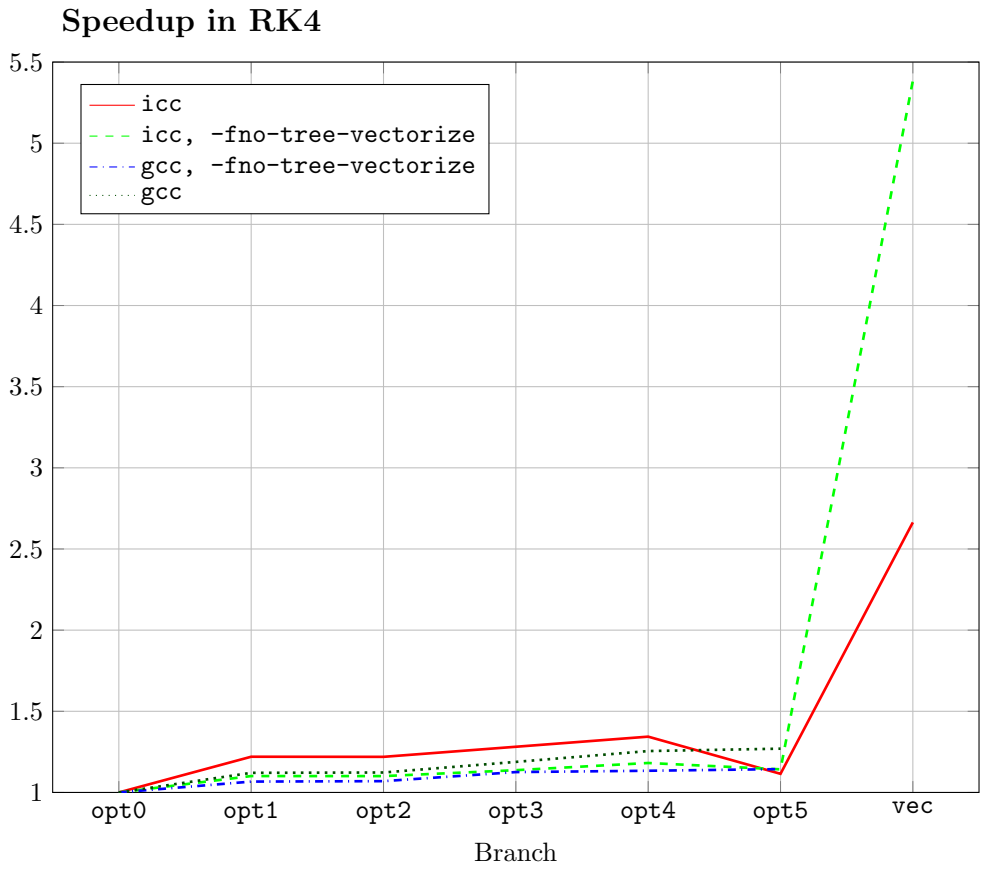

FIGURE 4.7: Speedup of `ParallelCyclotronTracker::rk4` and its subroutines over the baseline implementation.

4.3.2 The First Four Steps

In Figure 4.3, there is a definite downwards trend in the CPU time over the first four steps which motivates that the term optimization is actually justified. Furthermore, Figure 4.4 shows a steady climb in speedup over the baseline implementation towards 1.18× and 1.13× for `icc` and `gcc`, respectively. Overall, the differences between the compilers are fairly small in the big picture.

4.3.3 Inlining `ParallelCyclotronTracker::derivate`

For `opt5`, the inlining of all subroutine calls in `ParallelCyclotronTracker::rk4` as described in Section 3.6, `icc`, but not `gcc`, compiles slower code than before leading to a fairly large increase in computation time both inside (see Figure 4.6) and outside (see Figure 4.1) of `ParallelCyclotronTracker::rk4` which undoes a large part of the speedup gained in the previous optimizations. No obvious reason for this behavior is found during a quick investigation.

4.3.4 Vectorization[35]

Theoretically, a perfect vectorization would improve the performance of `ParallelCyclotronTracker::rk4` by an additional 4×. However, Figure 4.7 only shows a 2.4× speedup over the previous optimization `opt5`, which might lead us to believe that the vectorization is flawed. However, this does not take into account that the compiler might have already performed some vectorization by itself in the previous steps. We therefore disable auto-vectorization via the `-fno-tree-vectorize` flag and observe a speedup of around $\frac{5.39}{1.14} \approx 4.72\times$ between `opt5` and `vec` in Figure 4.7, which is even larger than what one would expect from perfect vectorization. This serves as an indication that the rather disappointing speedup previously observed had not been exclusively due to sub-optimal vectorization but to a large extent due to the compiler already vectorizing part of the code automatically. The fact that the latter speedup is even greater than four suggests that the vectorization step itself enabled further compiler optimizations in comparison to the previous stages. Comparing the CPU times, we can now also see that after vectorizing by hand it no longer seems to have a large impact whether auto-vectorization is enabled or not.

[35] As elaborated in Section 3.7.4.1 vectorization is only done for `icc`.

Chapter 5
Optimizing with Minimal Obfuscation

5.1 Idea

Having observed the massive code obfuscation introduced with the single core optimizations and the associated speedups, we note that such a trade-off does not seem worth it for this particular code. Therefore, we turn our attention towards a middle-way between aggressive application of *'compiler-friendly style'* and slow code by attempting to introduce only some of the optimization steps taken previously in a manner that requires the least intrusion in terms of readability.

5.1.1 Vectorization

We select the vectorization because it paid off the most.[36] While this step does alter the structure of the code quite a bit due to reordering, masking and shuffling, as mentioned in Section 3.7, these changes are easily documented and expected to be understood intuitively by the reader especially if they can be written in a self-documenting way. In the aggressively optimized code, we observe that compared to the already massively obfuscated code it is applied to, the vectorization step does not significantly deteriorate code legibility.

Thanks to the `intrinsics` overloading floating-point arithmetic operators, one can in many cases simply exchange the type of a floating point variable with its vector equivalent and keep the code performing operations on them unchanged as in Code 5.1.

```
/*
 * Returns true iff a real root of the polynomial p(x) = ax^2 + bx + c exists.
 * If such roots exist they are stored in x.
 */
bool quadraticFormula(double a, double b, double c, double x[2]) {
    double d = b*b - 4*a*c;
    if (d < 0) {
        return false;
    }
    else {
        x[0] = (-b + std::sqrt(d))/(2*a);
        x[1] = (-b - std::sqrt(d))/(2*a);
        return true;
    }
}

__m256d quadraticFormula(__m256d a, __m256d b, __m256d c, __m256d x[2]) {
    __m256d d = b*b - 4*a*c;

    __m256d flag = d < 0;
    d = _mm256_and_pd(d,flag); // avoiding domain error

    x[0] = (-b + _mm256_sqrt_pd(d))/(2*a);
    x[1] = (-b - _mm256_sqrt_pd(d))/(2*a);
    return flag;
}
```

Code 5.1: Computation of the real roots of a quadratic eqaution.

[36] See for example the difference between `opt5` and `vec` in Figure 4.4.

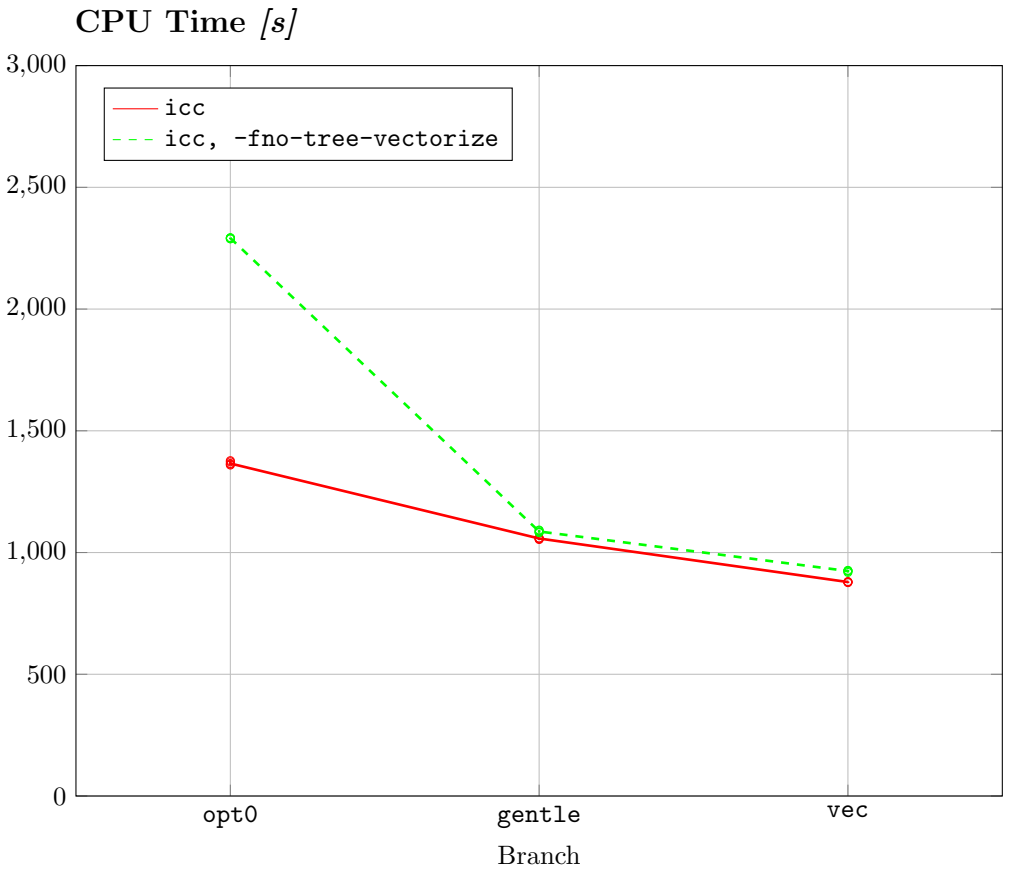

FIGURE 5.1: CPU time of the entire application.

5.1.2 Enum Factory

The second optimization we pick to implement in the less intrusive approach is the introduction of the `enum` factory `macro` pattern to replace the component type `std::string`s with `enum`s since that only introduces a minor change to code while providing a clear benefit.

5.1.3 Static Linking of `Cyclotron::apply`

As mentioned in Section 3.5 the call to `Cyclotron::apply` is virtual. In order to introduce a vectorized version of that virtual member function we wish to change the signature. Unfortunately, this means that we need to either change the base class such that we can keep dynamic linking or change the call to a static one. We choose the latter approach since this has already been accomplished in Section 3.5 and we are not required to introduce a dummy function to the base class but can keep our changes contained to the appropriate sections.

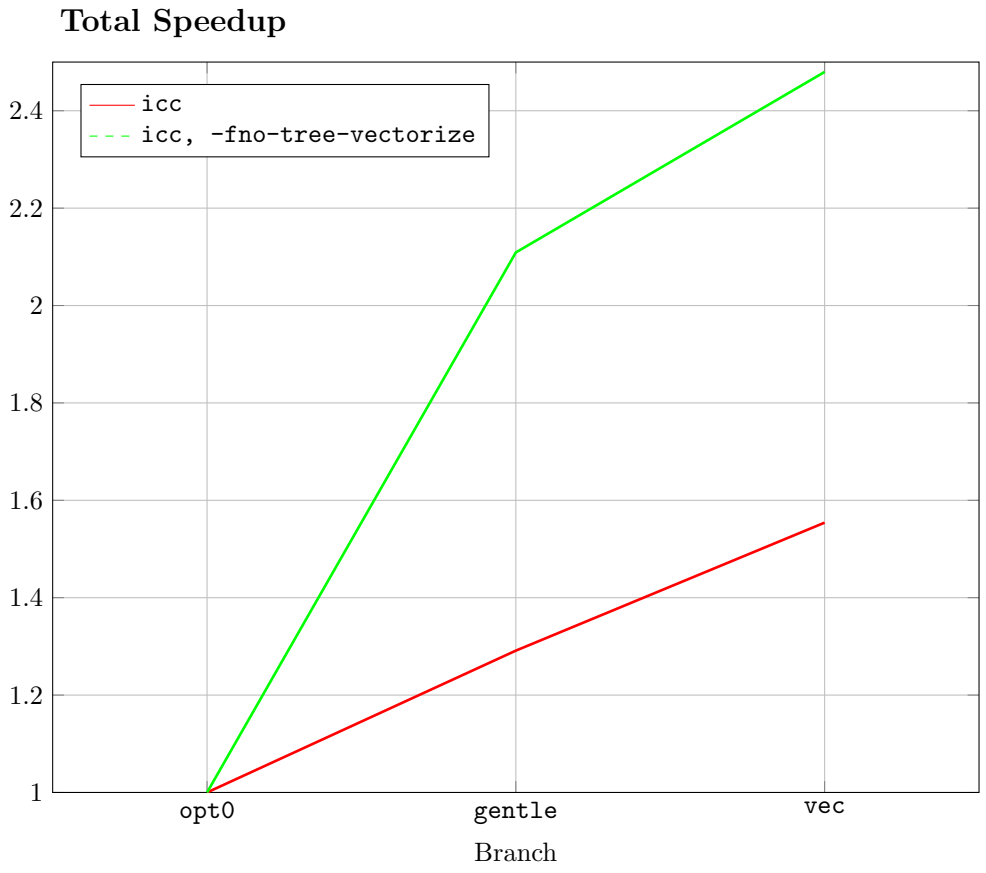

FIGURE 5.2: Speedup of the entire application over the baseline implementation.

5.2 Result

While this approach also incurs some of the complications presented in Section 3.7 – such as shuffling and masking – the final result resembles the base code much more closely.[37] The bulk of the computation looks fairly similar to the original code in that expressions and function signatures mostly only exchange the keyword `double` with `__mm256d`, no function calls are inlined and branches are simply replaced with easily documented masks.

Unfortunately, this lack of *'compiler-friendly style'* also has an impact on the performance. While one can observe in Figure 5.1 that the `gentle` vectorization closes the gap between the two compilation modes and in Figure 5.2 that a speedup of 1.29× is introduced, we still miss out on an additional 20%[38] improvement compared to the aggressively obfuscated code `vec`. The same picture also presents itself when we restrict our view to merely the hotspot in `ParallelCyclotronTracker::rk4` (see Figure 5.3), where it is additionally quite satisfying to observe that even without obfuscating the code very much, we can still achieve a speedup of well over 3.5× over the non-auto-vectorized code. This serves as a strong indication that even with only such 'gentle' changes the vector unit is being made better use of.

[37] For the code see branch `gentle` of the repository.

[38] In Figure 5.2, we observe that the speedup for the `vec` branch is around 1.55× which is roughly 120% of the 1.29× speedup observed for `gentle`.

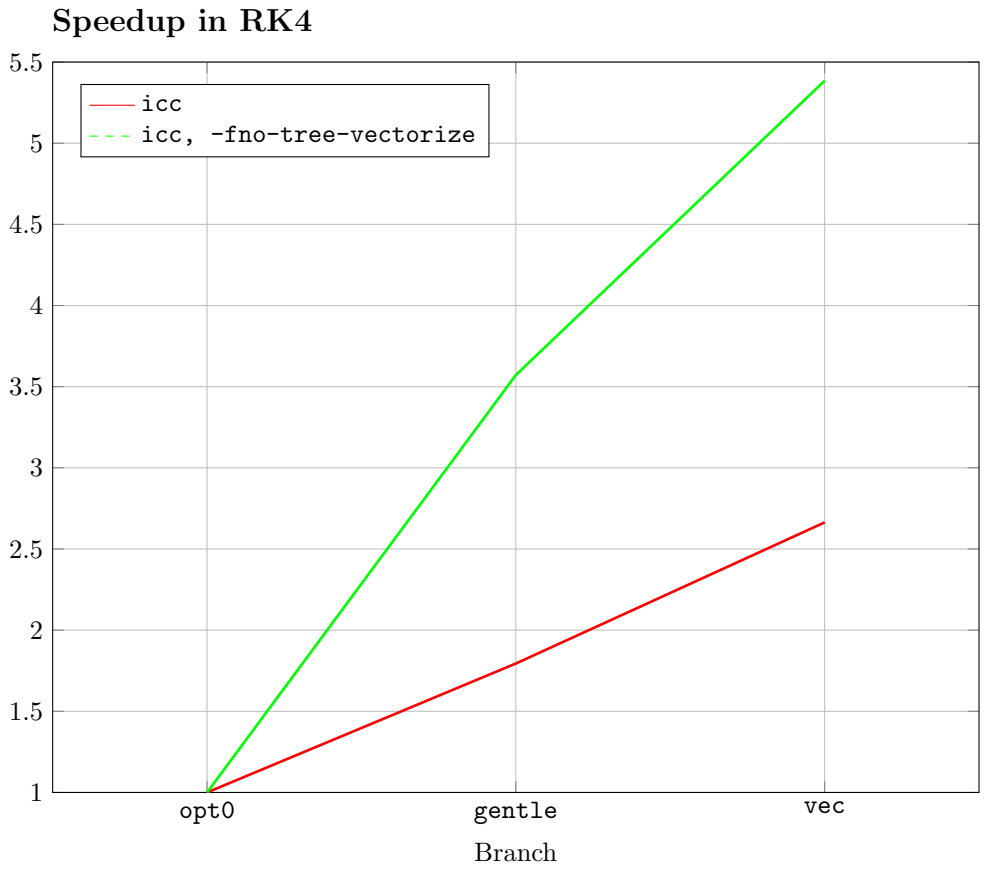

FIGURE 5.3: Speedup of `ParallelCyclotronTracker::rk4` and its subroutines over the baseline implementation.

5.3 Optimization Flags

Attempts to introduce some more compiler optimizations through setting flags which allow the compiler to optimize more aggressively – in particular following the guide at [9] and making use of [10] – as well as introducing the `restrict` keyword to remove the need to assume aliasing do not prove very fruitful, as the measured changes in runtimes across samples of code compiled with different flags is less than the variation among samples of the same executable.

Chapter 6
Conclusion

6.1 'Compiler-Friendly Style'

We are able to reduce the runtime of the hotspots in the `RingCyclotron` example of `OPAL` by applying the optimization techniques outlined in Chapter 2.3. However, this speedup comes at the unfortunate cost of rendering the code very hard to read and maintain despite documentation. On one hand, there is the issue that complex mathematical expressions now span a fairly large number of lines rather than being contained to just one line and that using single assignment makes it very cumbersome, if not impossible, to use meaningful variable names to write self-documenting code. On the other hand, the inlining bloats the code significantly. These drawbacks becomes more apparent the more code is present in the hotspots of the baseline implementation.

While it might be worth completely obfuscating a couple of lines of code to achieve an outstanding overall speedup, in this case we do not find it worth sacrificing maintainability of that much code in order to gain only such a humble overall speedup, especially considering that even the theoretically optimal overall speedup would not have been too satisfying.

6.2 Vectorization

Vectorization is of course a powerful way of improving performance even for a rather involved code as the one presented. Even without severely obfuscating the code a speedup of 1.79× could be achieved inside of the hotspots using vectorization. When we resort to sacrificing legibility of the code completely in favor of enabling compiler optimizations, we can even achieve a speedup of 2.66× in the relevant hotspot. The large discrepancy in runtime and speedup obtained by disabling auto-vectorization in the compiler options should also be noted as a sign that even for very involved code, where guided auto-vectorization fails, the compiler can nonetheless make decent use of the vector unit.

6.3 The Middle Way

Cherry-picking the optimizations that are least detrimental to the readability of the code turns out to be a rather nice approach. Without sacrificing too much maintainability we are able to provide some speedup by applying the global optimization of replacing string comparisons with integer comparisons where applicable and introducing vectorization in a way that disturbs the structure and look of the code only as little as needed. By forgoing *'compiler-friendly style'* we end up with fairly slower but still very readable code. We feel that this is the way to go for future projects with similar outsets.

6.4 Future Use

While this thesis provides both the gently vectorized code which only offers a speedup of 1.29× as well as the aggressively rewritten code which is roughly 1.55× faster than the original code, we would discourage working with the latter version in future projects, since, as a consequence of the rigorous rewriting in a more compiler-friendly manner, it has become highly error prone, as well as difficult to maintain and extend. It should therefore only be regarded as a proof of concept used to highlight the trade-off between beautiful and fast code. For readers seeking to build on an optimized version of the cyclotron simulation we suggest the more gently vectorized implementation complete with documentation of the changes where not self-documenting. Additionally, we would like to let this work serve as an example to guide future optimization and vectorization efforts away from aggressive obfuscation of the code in order to enable compiler optimizations and towards an approach similar to Chapter 5 if a similar ratio of lines of code to execution time percentage is observed.

Bibliography

[1] Andreas Adelmann, Achim Gsell, Tulin Kaman (PSI), Christof Metzger-Kraus (HZB), Yves Ineichen (IBM), Steve Russell (LANL), Chuan Wang, Jianjun Yang (CIAE), Suzanne Sheehy, Chris Rogers (RAL), and Daniel Winklehner (MIT). The OPAL (Object Oriented Parallel Accelerator Library) Framework. Technical Report PSI-PR-08-02, Paul Scherrer Institut, (2008-2014).

[2] AMAS Group (PSI). *AMAS Group webpage OPAL wiki*. https://amas.psi.ch/OPAL, retrived March 9, 2016.

[3] Using the gnu compiler collection (gcc). https://gcc.gnu.org/onlinedocs/gcc-4.5.3/gcc/i386-and-x86_002d64-Options.html, retrieved April 4, 2016.

[4] Markus Püeschel. *How To Write Fast Numerical Code* [lecture slides]. https://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring15/course.html, retrieved December 2015.

[5] Stackoverflow user "Suma". *Answer to Question "Easy way to use variables of enum types as string in C?" on stackoverflow.com*. http://stackoverflow.com/a/202511, 2008, retrieved March 9, 2016.

[6] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, July 2013.

[7] Intel. *Intel Intrinsics Guide*. https://software.intel.com/sites/landingpage/IntrinsicsGuide/#, retrieved March 9, 2016.

[8] Intel. *Intel C++ Compiler 16.0 User and Reference Guide*. https://software.intel.com/en-us/node/583200, retrieved March 9, 2016.

[9] Intel Developer Zone user "yang-wang (Intel)". *Step by Step Performance Optimization with Intel C++ Compiler*. https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler, October 9, 2014, retrieved March 9, 2016.

[10] Developer Products Division Software Solutions Group Intel Corporation. *Floating-point control in the Intel compiler and libraries or Why doesnt my application always give the expected answer?* [presentation slides]. https://software.intel.com/sites/default/files/article/326703/fp-control-2012-08.pdf, 2012, retrieved March 9, 2016.

Appendix A
VTune Amplifier Scripts

The following is a collection of the makeshift bash functions defined in order to run multiple hotspot analyses on the same program version and extract the desired information from the command line version of `VTune Amplifier`. These are used to collect the measurements for the figures in Chapters 4 and 5. There is also a GUI version of `VTune Amplifier` which is very well suited for interactive exploration of the measurements.

```
# Appends time CPU time and CPU usage of most recent collection to first argument.
append_time() {
    amplxe-cl -report summary | grep CPU | tail -n 2 | sed 's/[^0-9.]//g' | tr '\n' ',' | sed 's/,$/\n/g' >> $1
}

# Appends rk4 percentage of most recent collection to first argument.
append_percentage() {
    amplxe-cl -report top-down | grep "rk4" | sed 's/[^0-9.%]//g' | sed 's/%.*/%/g' | sed 's/^4//g' | tr '\n' ','
     >> $1
}

# Collects percentage and cpu time for multiple executions.
collect_hs() {
    touch $2;
    for i in `seq 1 $1`;
    do
        amplxe-cl -collect hotspots mpirun -np 8 src/opal RingCyclotron.in;
        append_percentage $2;
        append_time $2;
    done
}

# Gets cpu times of top hotspots.
get_hs() {
        amplxe-cl -report hotspots | tr -s ' ' | sed 's/\([^ ]*\) [^ ]* \([^ ]*\).*/\1\t\2/g' | head > $1
}
```

Appendix B
RingCyclotron.in


```
Option, ECHO=FALSE;
Option, ENABLEHDF5=FALSE;
Option, PSDUMPFREQ=100000;
Option, SPTDUMPFREQ = 10;
Option, PSDUMPEACHTURN=false;
Option, PSDUMPLOCALFRAME=FALSE;
Option, REPARTFREQ=20;
Option, ECHO=FALSE;
Option, STATDUMPFREQ=100000;
Option, CZERO=FALSE;
Option, TELL=TRUE;
Title,string="OPAL-cycl: the first turn acceleration in PSI 590MeV Ring";

Edes=.072;
gamma=(Edes+PMASS)/PMASS;
beta=sqrt(1-(1/gamma^2));
gambet=gamma*beta;
P0 = gamma*beta*PMASS;
brho = (PMASS*1.0e9*gambet) / CLIGHT;

//value,{gamma,brho,Edes,beta,gambet};

phi01=139.4281;
phi02=phi01+180.0;
phi04=phi01;
phi05=phi01+180.0;
phi03=phi01+10.0;

volt1st=0.9;
volt3rd=0.9*4.0*0.112;

turns = 1;
//nstep=2000;
nstep=500;
//nstep=5;

frequency=50.650;
frequency3=3.0*frequency;

ring: Cyclotron, TYPE="RING", CYHARMON=6, PHIINIT=0.0,
PRINIT=-0.000174, RINIT=2130.0, SYMMETRY=8.0, RFFREQ=frequency,
FMAPFN="s03av.nar";

rf0: RFCavity, VOLT=volt1st, FMAPFN="Cav1.dat", TYPE="SINGLEGAP",
FREQ=frequency, RMIN = 1900.0, RMAX = 4500.0, ANGLE=35.0,  PDIS = 416.0,
GAPWIDTH = 220.0, PHI0=phi01;

rf1: RFCavity, VOLT=volt1st, FMAPFN="Cav1.dat", TYPE="SINGLEGAP",
FREQ=frequency, RMIN = 1900.0, RMAX = 4500.0, ANGLE=125.0, PDIS = 416.0,
GAPWIDTH = 220.0, PHI0=phi02;

rf2: RFCavity, VOLT=volt3rd, FMAPFN="Cav3.dat", TYPE="SINGLEGAP",
```

```
FREQ=frequency3,RMIN = 1900.0, RMAX = 4500.0, ANGLE=170.0, PDIS = 452.0,
GAPWIDTH = 250.0, PHI0=phi03;

rf3: RFCavity, VOLT=volt1st, FMAPFN="Cav1.dat", TYPE="SINGLEGAP",
FREQ=frequency, RMIN = 1900.0, RMAX = 4500.0, ANGLE=215.0, PDIS = 416.0,
GAPWIDTH = 220.0, PHI0=phi04;

rf4: RFCavity, VOLT=volt1st, FMAPFN="Cav1.dat", TYPE="SINGLEGAP",
FREQ=frequency, RMIN = 1900.0, RMAX = 4500.0, ANGLE=305.0, PDIS = 416.0,
GAPWIDTH = 220.0, PHI0=phi05;

l1:   Line = (ring,rf0,rf1,rf2,rf3,rf4);

Dist1:DISTRIBUTION, DISTRIBUTION=gauss,
            sigmax = 2.0e-03,
            sigmapx = 1.0e-7,
            corrx = 0.0,
            sigmay = 2.0e-03,
            sigmapy = 1.0e-7,
            corry = 0.0,
            sigmat = 2.0e-03,
            sigmapt = 3.394e-4,
            corrt=0.0;

Fs1:FIELDSOLVER, FSTYPE=FFT, MX=32, MY=32, MT=16,
         PARFFTX=true, PARFFTY=true, PARFFTT=true,
         BCFFTX=open, BCFFTY=open, BCFFTT=open, BBOXINCR=2;

beam1: BEAM, PARTICLE=PROTON, pc=P0, SPACECHARGE=true, NPART=1E6, BCURRENT=1.0E-3, CHARGE=1.0, BFREQ= frequency;

Select, Line=l1;

TRACK,LINE=l1, BEAM=beam1, MAXSTEPS=nstep*turns, STEPSPERTURN=nstep,TIMEINTEGRATOR="RK-4";
 run, method = "CYCLOTRON-T", beam=beam1, fieldsolver=Fs1, distribution=Dist1;
endtrack;

Stop;
```

Appendix C
Machine Information

The following is the first of eight blocks of the contents of `/proc/cpuinfo` of the machine used. Apart from the processor number and `id`s the information is the same for the other seven blocks.

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 62
model name      : Intel(R) Xeon(R) CPU E5-2609 v2 @ 2.50GHz
stepping        : 4
microcode       : 1046
cpu MHz         : 2500.000
cache size      : 10240 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx
    fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good xtopology
    nonstop_tsc aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid dca
    sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm arat epb xsaveopt pln pts dts
     tpr_shadow vnmi flexpriority ept vpid fsgsbase smep erms
bogomips        : 4987.93
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
```