JIT-COMPILER FOR DYNAMICALLY CONTROLLED PARTICLE SYSTEMS

Benjamin Flück, Jakob Progsch, Simon Laube

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

We explore the application of a domain specific Just-In-Time (JIT) compiler to programmable particle systems and compare the resulting performance with a conventionally optimized C implementation. We show that building a JIT compiler for our sufficiently narrow scope can match and even exceed the performance of the C implementation, while lessening the optimization burden on the programmer.

1. INTRODUCTION

Traditionally environmental animations for games and movies were done procedurally or by hand. With the ongoing desire to produce more realistic and detailed environments, these animations are becoming too numerous and complex. Thus they become too costly to be done procedurally or by hand. The steady increase in available compute power in the last few years made it possible to replace these time and cost intensive task with small simulations that produce physically convincing results while still providing a degree of influence to the creator.

Motivation. In the light of these requirements and circumstances we decided to look at rule based particle simulations. A given set of rules is applied to the particles in each time-step. These simulations are especially interesting as they allow to model a wide range of environments depending on the set of rules governing a given scenario. As the number and composition of such rules is arbitrarily large and can dynamically change over the course of the simulation, it is hard to produce optimized code for such a simulation. We have therefore decided to explore the possibility of creating a Just-In-Time compiler (JIT) specialized for such particle simulations.

Limiting the scope of the compiler input to a well defined and limited problem allows our JIT compiler to incorporate a lot of domain specific knowledge and optimizations directly into its inner workings. For example, our JIT compiler knows the data layout for the particles and does not have to support any number of possible container or data formats.

The authors thank Alen Stojanov and Daniele Spampinato for their input during this project

Related work. JIT compilers are located somewhere between traditional Ahead-Of-Time compilers (AOT) such as clang[1], gcc[2] or icc[3] and interpreters for scripting languages such as JavaScript. As such they often include techniques from both end of this spectrum. The core idea is that code is only translated to machine instructions at the latest possible time. This makes JIT compilers well suited for dynamic languages such as JavaScript or Lua as well as situations in which information about the target architecture is either not available beforehand or can be used to increase performance. JIT compilers are widely used, most notably in the form of the Java Virtual Machine[4] and the JavaScript[5][6] engines found in web browsers. Another state of the art JIT compiler from which we took some inspiration for the internal code represenation is LuaJIT [7][8], a JIT compiler for the Lua language. Other uses of JIT compilers that resemble our use case are found in graphics card drivers to translate generic shader code (OpenGL) and compute kernels (OpenCL) to highly optimized hardware specific machine code.

2. IDEA AND BACKGROUND

In this section we provide a detailed description of the particle system we use, the domain specific language (DSL) the JIT compiler accepts as input, how the JIT compiler is structured, and conduct a cost and complexity analysis.

Particle System. The particle simulation we concentrate on consists of two main components, the particles themselves and the set of rules governing the interactions and progress of the simulation. The particles are provided in the following way:

```
struct particle_t {
    float position[3];
    float mass;
    float velocity[3];
    float charge;
} particle;
```

This allows for 16 bytes aligned access to both the position and velocity vectors, as well as creating rules for diverse application, e.g. for physical or chemical simulations.

As for the rules, they are provided as functions that take a particle and an array of rule specific values as input. This function then computes and applies the rule specific update to the given particle. This setup allows for a great flexibility on what the rules can do, and even the time step for the simulation itself is such a rule:

```
newton_step_apply(particle *p, void *d){
    float dt =  d[0];
   p->position[0] += dt*p->velocity[0];
   p->position[1] += dt*p->velocity[1];
   p->position[2] += dt*p->velocity[2];
}
```

In the end the complete simulation follows three basic steps:

```
(1) get particles
(2) get rules
(3) iterate over all particles
    and apply every rule to them
```

If the environment changes between iterations one has to update the list of rules to adapt the simulation to the new circumstances. The simulation can then proceed immediately afterwards.

DSL. To conveniently define the rules for the JIT compiler we introduced a domain specific language (DSL). The DSL provides a subset of C operator syntax including arithmetic, bitwise and ternary operators, temporary float variables and array access on the input arrays. This feature set is sufficiently expressive for the intended purposes while still being easy and fast to parse.

JIT Compiler. In contrast to the statically optimized code, the JIT compiler exploits possible optimizations across multiple rules by fusing all the rules into one executable function. To translate the DSL into its intermediate representation, a straight forward hand written single pass lexer and recursive descent parser was written.

For the intermediate representation a static single assignment form representation (SSA-IR)[9, Chapter 6.2.4][8] is used since it allows efficient implementation of the relevant optimization passes. Because the DSL does not contain flow control constructs, the optimizer and code generator only ever deal with a single basic block. This significantly reduces the analysis that is required to optimize the code as compared to a general purpose compiler. The last step of translating the SSA-IR to machine code is performed using a x86-64 instruction encoder (PLASM). The code generator assigns registers and stack spilling locations on the fly and emits VEX encoded instructions only. The VEX encoding allows the use of non destructive instructions which are very close to their SSA-IR counterparts. To make the code actually executable it is wrapped in a function template and written to memory, which is then switched to executable state by means of a system call.

Cost Analysis. While it is possible to determine the exact op-count for any given rule, the final op-count depends on the number and nature of the involved rules and can change throughout the simulation. Therefore there is no global cost measure. The op-count relative to the input size, in our case number of particles, is always $O(1)$. For all except the smallest simulations including only very few rules, the neglected constant factor pushes our computation far into the compute bound region. Furthermore our simulation exhibits perfect spatial locality as it passes over the particles linearly. Given that our JIT compiler changes the number of stores, loads, and operations required for applying the rules, it is not possible to simply compare the performance in flops/cycle between our jited code and static optimized code. The reason simply being that code with a lower performance but requiring fewer operations may still be faster than code requiring more operations and achieving higher operational intensity.

We have therefore decided to look at the runtime, more precisely at the cycles per particle. This allows for a precise comparison how our jited code performs in comparison to the conventional code in an absolute way.

3. IMPLEMENTATION DETAILS

Starting from a simple, straight-forward baseline implementation we have introduced two main paths of optimization which we discuss in detail in this section. On one hand we applied conventional optimizations i.e. static optimizations that are applied at compile time, to the baseline implementation. On the other hand we have the JIT compiler itself, which implements all the necessary steps to dynamically produce executable code at runtime.

Baseline Implementation. Our baseline version implements a number of rules, such as gravitation or collisions, each accessible through a function pointer, and an array of particles. In order to run the simulation one can now simply collect a list of desired rules and pass them to the main simulation function. This function then simply iterates over all particles and applies every rule to each one of them.

Conventional Optimizations. The dynamic nature of the rule composition limits a conventional compiler or the programmer to optimizing the code on a per rule basis. With the baseline code already written in SSA form (static single assignment) we have then implemented two variants of vectorization. The first one implementing 3D-vector arithmetic in SSE code and the second one processing multiple particles in parallel.

Implementing 3D-vector arithmetic may seem attractive but poses two problems. SSE registers can hold four floats (single precision floating point number) in 128 bits which means for 3D-vectors we already waste 1/4th of the register space and going to the newer AVX instruction at 256 bit wide registers means either wasting 5/8th of the available space or rewriting all the code to process two 3D-vectors in parallel. Furthermore there are no dedicated instructions to calculate the euclidean length of a vector and therefore require significant overhead for data shuffling. The one advantage of this approach lies in the treatment of conditional code, e.g. in collision detections. As only one particle is processed at a time it is possible to skip unneeded computations.

Processing multiple particles by packing their respective x, y, and z coordinates into different register reverses these drawbacks and advantages. This treatment allows to go from SSE to AVX instructions by simply loading twice as many particles into the registers. In contrast it is no longer possible to use conditional statements to skip portions of the code. A conditional check may be true for some particles and false for others. Instead we use the conditional check to produce a mask indicating for which particles the check holds and for which it doesn't. We then run the conditional code for all particles and mask the application of the results according to this indication mask.

JIT Compiler. The JIT compiler exclusively uses the multiple particle approach for vectorization which allows the intermediate (SSA-IR) code to only deal with the scalar form of the rules that are then executed on 1, 4 or 8 particles at a time. This means vectorization exclusively happens in final code generation and is of no concern for the optimization passes implemented on the SSA-IR form.

The rules are initially given as strings in a strongly reduced C style syntax. For example the euler update rule looks as follows:

```
dt = [8]
[0] = [0] + dt*[4]
[1] = [1] + dt*[5]
[2] = [2] + dt*[6]
```

The numbers in brackets are indexing into implicit arrays that have their meaning assigned by the code generator. In our case indices 0-7 refer to particle components and higher indices refer to the rule specific input values.

These rules are then translated to SSA-IR form by a single pass recursive descent parser. The SSA-IR form is stored as an array of fixed width 64bit instructions which contain an opcode, flags and up to three operands. The operands are indices that either refer to previous instructions in the SSA-IR array or to the input arrays in case of the load and store instructions. Using this flat and fixed width representation allows efficient implementation of optimizations passes.

Optimization. The first optimization pass consists of removing redundant stores and loads. Since concatenating rules will result in multiple redundant loads (almost every rule will load position for example) and stores (only the last store to any component will actually have an effect on the memory), a significant amount of those operations can be combined or removed. Similarly, redundant computations which show up as identical instructions in the SSA-IR can also be skipped. Both of these operations are done by updating an index remapping array. All uses of redundant instructions are remapped to use the first instance of that instruction. As a result the additional instances are not referred to anymore and left as dead code.
In the next step instructions are recursively marked as being live starting from the store instructions. The live instruction can then be compacted again removing all the dead code including the redundant operations from the previous steps. Depending on the rule combination this process can remove up to half the instructions of the original rules. However, it does not reduce the length of the critical path.
The last step before code generation is the scheduler. The scheduler tries to heuristically reorder the instructions into a better sequence. Since the processors we are targeting have reasonably large reorder buffers (168 slots for Intel Ivy Bridge microarchitecture) the exact ordering between single instructions is not that important. Still we want largely independent paths of execution to be interleaved in the greater scheme since the loop bodies as a whole easily exceed the reorder buffer size. Additionally the live ranges of values can be changed by rescheduling, which allows the scheduler to influence the register pressure the code generator will have to deal with. The scheduler therefore operates by enumerating all eligible instructions that have their dependencies fulfilled and assigns a score to them. It then schedules the instruction with the highest score and repeats the process until all instructions are scheduled.
There are two factors that influence the score: criticality and influence on live registers. Criticality is the sum of latencies of dependent instructions. Influence on live registers is the change in the amount of live registers the instruction results in when scheduled. The scoring therefore prefers instructions that have high criticality and reduce the amount of live registers.

Code Generation. Each of the three vectorization levels (scalar, 4-wide, 8-wide) have their own code generator. The main difference between the versions is how they translate load and store instructions since the non-scalar versions need to gather the particle components from memory into registers. Instruction selection is done by a lookup table since the SSA-IR instructions have already been chosen to directly correspond to the underlying x86-64 instruction set. The main challenge therefore is the allocation and spilling of registers. To utilize the instruction sets ability to directly use memory operands for arithmetic instructions, load operations are deferred to the first usage of their result. When spilling is necessary, the register value that has its next use the farthest from the current position is assigned a free slot on the stack and moved there. In addition to the translation of the block itself, the code generator also generates some fixed code such as the loop structure and function boilerplate around the optimized rules, to make them a callable function following the system-V AMD64 ABI. This way the resulting buffer can simply be cast to an appropriate function type and called from the C code. The code generator allocates memory using the `mmap` system call to obtain page aligned memory with read and write permissions. Before returning, that memory is changed to read and execute permissions using the `mprotect` system call. This is necessary since memory obtained from `mmap` will typically not have execute permissions and calling into it would therefore result in a segmentation fault.

4. EXPERIMENTAL RESULTS

In this section we briefly describe our experimental setup, then provide some performance data on the JIT compiler itself, i.e. how fast it compiles, and finally we compare the code produced from the JIT compiler with the conventionally optimized code.

Experimental setup. All measurements were performed on a Intel Core i7-3632QM (Ivy Bridge) processor running at a clock frequency of 3.2 GHz with cache sizes of 32kB (L1 data), 256kB (L2) and 6MB (L3). The clang compiler version 3.5 was used for all measurements with optimization flags: `-O3 -march=native`.

For the particle system itself we have seen a nearly non-existing influence from the number of particles. When going from a small particle set that fits into the first level cache to a multi GB set the effect on the overall runtime is around one or two additional cpu cycles per particle and therefore can simply be ignored for any practical purpose. This behaviour is explained due to the perfect sequential traversal over the particles and the high enough computational intensity. This provides a perfect data access pattern for the hardware prefetcher to mask the memory accesses.

JIT Compiler. First we consider the performance of the JIT compiler itself. While compilation speed was not our primary concern, we still have to make sure it is suited for real time usage. Meaning we want compilation times to be at most in the low milisecond range. For reference, the time budget to calculate a single update cycle in real time graphics tasks is typically 8-30 miliseconds.

If we take a look at where the jit compiler spends its time (fig 1) we immediately see that all except two stages run in the range of a few microseconds and only the scheduling and code generation take significantly more time.
The code generation runs in approximately constant time, around 40 to 120 microseconds, where most of the time is used for two system-calls. These two calls, `mmap` and `mprotect`, cause page faults and context switches within the operating system and therefore impose a constant cost that is hard to avoid. The actual writing of the code is only a minor part of the generation stage.
Turning our attention to the scheduling stage, we immediately see that the time used for the scheduling step is strongly dependant on the number of rules, and by extension the number of instructions that are composed together. For scenarios with a few rules, the scheduling is eclipsed by the code generation, but for more complex scenarios scheduling takes more time. Still, the whole JIT compiler runs in the sub-millisecond range for reasonable use cases and is therefore fast enough for real time usage.

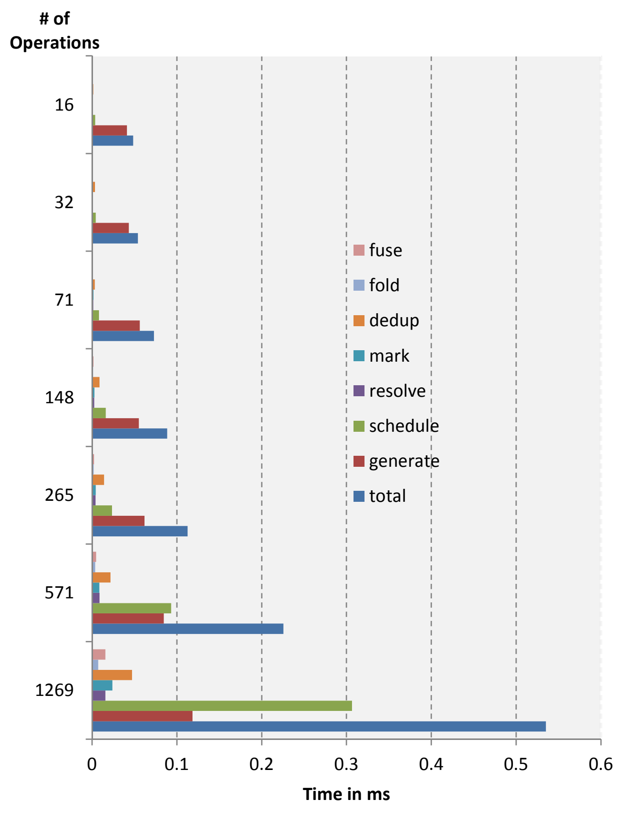

Fig. 1. Performance of the JIT compiler itself

Code Perfomance. When looking at the performance of the code itself, we distinguish two situations. The first one encompasses scenarios were there is only one or two rules, which allows us to see how well our JIT compiler performs in direct comparison with AOT compiler. The second scenario takes a closer look at simulations with at least four and up to more than twenty rules, exhibiting the potential for optimizations across multiple rules. The results in the plot represent the following configurations:

naive: the naive implementation in SSA form

vector SSE: implemented 3D-vector math in SSE instructions

parallel SSE 1: parallel processing of four particles without loop unrolling

parallel SSE 8: parallel processing of four particles with further 8-fold loop unrolling

JIT scalar: the naive implementation as produced by the JIT compiler by emitting scalar operations only

JIT AVX 4: vectorized code form the JIT compiler using 128bit vector register, i.e processing 4 particles in parallel

JIT AVX 8: vectorized code from the JIT compiler using the extended 256bit register, i.e processing 8 particles in parallel

Looking at figure 2 we can see that both our JIT compiled and hand optimized code outperform the naive implementation to varying degrees. Due to the required data shuffling neither can fully realize the theoretical speed-up from vector instructions. In general we see the JIT compiled code to be at least equally as fast as the conventional code. Also note that going from 4-way to 8-way vector instructions does not yield significant speed ups and in some cases even results in slow downs. This may be in part caused by added data shuffling. On the used processor, some instructions (`sqrtps` and `divps` specifically) also have twice the latency and half the throughput in their 256bit versions. Therefore they do not provide an actual advantage over their 128bit counterparts. The test cases called *central4* and *central8* in figure 3 are strongly affected by this, since their running time is dominated by those two instructions.

The test cases shown in figure 3 use more rules at once and we see the results shift in favour of both optimization paths, with an advantage for the JIT compiler. For both the JIT compiled and conventional code the presence of many rules and therefore many operations allows to compensate the data shuffling overhead we have seen in the previous comparison. Looking closely at the results we can see that the `jit avx 8` code always outperforms every other version, and the `jit avx 4` code outperfoming the equivalent hand optimized code, `parallel sse 1`, as well. Comparing the two conventionally optimized code versions `parallel sse 1` and `parallel sse 8` indicates that loop unrolling further increases instruction level parallelism, since the version without loop unrolling is still latency limited.

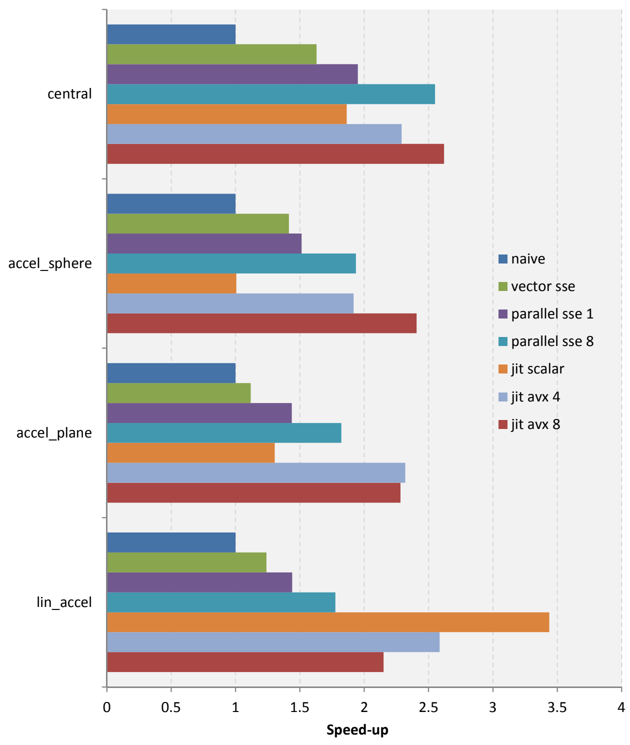

Fig. 2. Performance comparison for single or dual rule scenarios

One of the main advantages of the JIT compiler is that it significantly reduces the amount of actually generated instructions. The optimization passes eliminate up to half of the instructions present in the input rules. Due to the types of optimizations that are performed it does however not reduce the critical path length. So the total latency of one loop iteration is unaffected when assuming perfect instruction level paralellism. This reinforces the observation that loop unrolling could provide a further speed up for the JIT compiled code by allowing more efficient interleaving of independent paths of execution.

5. CONCLUSIONS

In this paper we have introduced the idea of using a Just In Time compiler for creating simulations of dynamic particle systems. We have then outlined how such a purpose built JIT compiler can take advantage of implicit knowledge and simplified programming requirements. Furthermore we

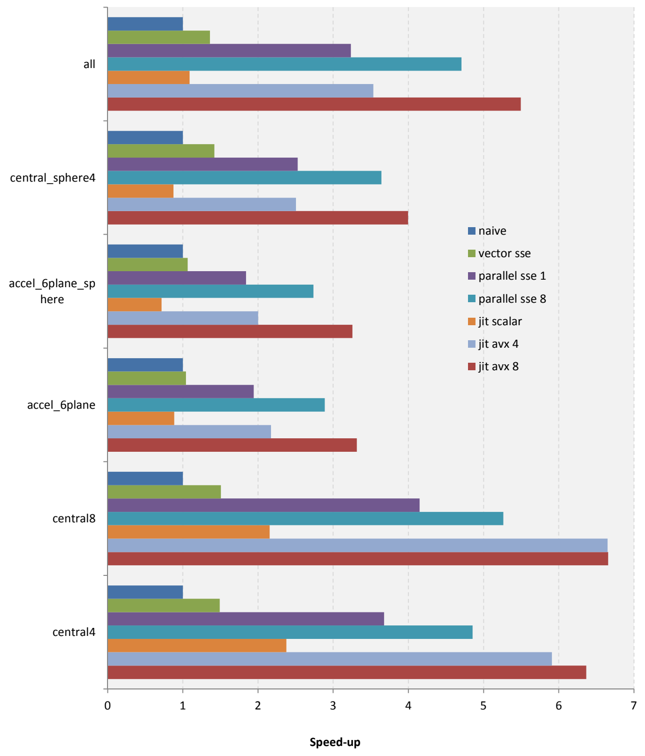

Fig. 3. Performance comparison for multi rule scenarios

have shown that even a simple implementation can already produce code that is performing at least equivalently or better than conventionally optimized code. Besides direct advantages in performance, having such a JIT compiler also relieves the programmer from having to hand optimize every new simulation rule he might come up with. After the initial effort to create such a JIT compiler its usage provides the best bang for buck for the programmer, he gets the performance of heavily optimized code at the cost of a simple and straight forward implementation. Originally both the conventional code and the JIT compiler were producing vectorized code using 128bit wide register with the difference that the JIT compiler only needed little work and changes in the backend to produce new AVX code, while it would be necessary to go over all existing code and redo all the work for the conventional code. This further highlights the advantages of building such an infrastructure, not only to gain immediate performance gains, but also savings for future adjustments when they come along.

Despite the limited scope of this short semester project, we managed to show the value and possible gains of a domain specific JIT compiler, by implementing just a small subset of the optimizations conventional compilers deploy. In a larger context this also furthers the view that automatic code generation and optimization are well worth the effort.

Future Work. We have already highlighted the possibility of further loop unrolling which allowed the conventional code to partially close the gap to our JIT compiled code. Other possible optimizations not currently implemented include performing algebraic simplifications on the IR and substituting expensive operations such as square roots with fast reverse square root and newton approximations. Another interesting aspect of the online code generation is the possibility to instrument code at runtime and refine heavily used kernels by performing essentially profile guided optimizations for the specific machine the code is currently running on.

6. REFERENCES

[1] "clang: a c language family frontend for llvm," `http://clang.llvm.org/`.

[2] "Gcc, the gnu compiler collection," `http://gcc.gnu.org/`.

[3] "Intel compilers," `https://software.intel.com/en-us/intel-compilers`.

[4] "Openjdk," `http://openjdk.java.net/`.

[5] "V8 javascript engine," `http://code.google.com/p/v8/`.

[6] "Spidermonkey javascript engine," `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`.

[7] Mike Pall, "Luajit," `http://luajit.org/`.

[8] Mike Pall, "Luajit 2.0 ssa ir," `http://wiki.luajit.org/SSA-IR-2.0`.

[9] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques and Tools*, Paerson, 2014.