Extending LLVM for Lightweight SPMD Vectorization: Using SIMD and Vector Instructions Easily from Any Language

Robin Kruppe, Julian Oppermann, Lukas Sommer, Andreas Koch
Embedded Systems and Applications Group, TU Darmstadt, Germany
{kruppe,oppermann,sommer,koch}@esa.tu-darmstadt.de

***Abstract*—Popular language extensions for parallel programming such as OpenMP or CUDA require considerable compiler support and runtime libraries and are therefore only available for a few programming languages and/or targets. We present an approach to vectorizing kernels written in an existing general-purpose language that requires minimal changes to compiler front-ends. Programmers annotate parallel (SPMD) code regions with a few intrinsic functions, which then guide an ordinary automatic vectorization algorithm. This mechanism allows programming SIMD and vector processors effectively while avoiding much of the implementation complexity of more comprehensive and powerful approaches to parallel programming. Our prototype implementation, based on a custom vectorization pass in LLVM, is integrated into C, C++ and Rust compilers using only 29–37 lines of frontend-specific code each.**

I. Introduction

CUDA demonstrated the advantages of extending an existing general-purpose language such as C++ with the ability to seamlessly integrate serial code and parallel kernels written in Single Program Multiple Data (SPMD) style, rather than using a separate language for these kernels. However, CUDA is not portable, especially not to current and future masked-SIMD instruction sets, which lend themselves to a similar programming style as demonstrated by ispc [1].

But even more portable parallel programming environments such as OpenMP still require considerable compiler support and runtime libraries and are therefore only available for a few programming languages. Ideally, programs written in many different serial languages could enable parallel processing through far smaller, more targeted language extensions.

The ability to spawn and manage multiple threads as well as communication between them is already widespread today, so a minimalist approach may delegate these aspects to user code rather than baking them into the language or compiler.

On the other hand, exploiting the SIMD or vector instructions of modern processors is also crucial for achieving high performance, but manual access to these instructions from general purpose languages is often non-existing, limited or overly target-specific, while fully automatic vectorization is severely restricted by the need to preserve serial semantics.

We therefore propose to limit the involvement of the compiler to automatically vectorizing SPMD-style kernels at the request of the programmer, rather than providing a fully-featured parallel programming environment. Specifically, we describe a small set of intrinsic functions that can be used by programmers to annotate (SPMD) code regions. These code regions can then be vectorized by established approaches whose implementation can be contained entirely in a shared compiler middle end such as LLVM.

Thus all that is required to port this approach to more languages is to expose the intrinsic functions to the user and translate them to matching intrinsics in the optimizer IR, which is typically trivial as such intrinsics are commonplace.

II. A Simple Example

To double each element of an array with `n` elements in C, a programmer may write an SPMD kernel as follows:

```
void add_kernel(int n, float *x) {
  size_t i = lane_id();
  if (i > n) return;
  a[i] *= 2.0;
}
```

When this function is vectorized, it processes as many elements at once as there are SIMD lanes on the target processor, as the intrinsic `lane_id()` will be mapped to the index vector `<0, 1, 2, 3, ...>`.

The programmer then writes code to schedule multiple kernel invocations to cover the entire problem instance. The best way to do this depends on the task at hand, but in this example, a single-threaded implementation may be as simple as the following loop:

```
for (size_t j = 0; j < n;
     j += /* vector width */;) {
  spmd_call(add_kernel, n, &x[j]);
}
```

The use of the `spmd_call` intrinsic indicates to the compiler that the function `add_kernel` should be vectorized and the vectorized version should be called with the given arguments replicated into all SIMD lanes. It is still possible to call the unvectorized function directly, and thus to share code (and data) between parallel and serial parts of the program.

While repeating this boilerplate code for every instance of parallel processing would be unacceptable, it is easily possible to extract common patterns into library code without modifying and rebuilding the compiler or compiler-provided

runtime library. For example, a Rust programmer with access to the aforementioned intrinsics can write a small library enabling them to express the above example as follows at no loss of performance:

```
parallel_for_each(x, |x_element| {
  x_element *= 2.0;
});
```

Thus our approach shifts the power and responsibility of creating higher-level or domain-specific abstractions for data parallelism from the compiler to the programmer. This fills a gap between purely serial or multi-threaded programming without access to SIMD instructions on the one hand, and more powerful and more heavyweight parallel extensions such as CUDA or OpenMP on the other hand.

III. Evaluation

We prototyped this concept by implementing a custom vectorization pass for LLVM based on previous research such as Whole Function Vectorization [2] which vectorized OpenCL kernels for CPUs. Our pass operates on LLVM IR and emits the portable vector operations provided by LLVM, so it could conceivably be used to target many different architectures, although we have only evaluated it on one.

The pass can be plugged into the optimization pipeline of any LLVM-based compiler, which we have done for C, C++ and Rust compilers. This integration, together with adding our intrinsics, required only 29–37 lines of code per frontend.

Targeting the open GPGPU processor Nyuzi [3] and its 16-wide masked-SIMD unit, we achieve speedups of up to 10x over scalar code and are competitive with some manually vectorized code.

The evaluation also confirms the utility of masked-SIMD instruction sets: despite making no effort to exploit uniformity of computations and control flow across SIMD lanes, our implementation generates well-vectorized code and avoids previously observed slowdowns [2] on CPU architectures without masked-gather and masked-scatter instructions.

IV. Future Work

In the future we hope to combine this approach with other implementations of vectorization algorithms such as RV [4] and apply it to other architectures such as Intel's AVX-512, Arm's Scalable Vector Extension, or RISC-V's upcoming vector extension.

References

[1] M. Pharr and W. R. Mark, "ispc: A SPMD compiler for high-performance CPU programming," in *Innovative Parallel Computing*, 2012.

[2] R. Karrenberg and S. Hack, "Whole Function Vectorization," in *International Symposium on Code Generation and Optimization*, 2011.

[3] [Online]. Available: https://github.com/jbush001/NyuziProcessor/

[4] [Online]. Available: https://github.com/cdl-saarland/rv