llsimd: Portable SIMD intrinsics through LLVM IR

Bhavjit Chauhan and Anton Kazachenko
Simon Fraser University

Introduction

Single instruction, multiple data (SIMD) is a form of parallel processing on the CPU. Desktop processors first adopted SIMD as we know it in the 1990s. Since then, chipmakers have spent more and more chip area for this form of computing and most of the new instructions additions are SIMD instructions. Despite their established foundation, however, these instructions do not have a standardized interface. C++ has only recently begun introducing the experimental SIMD library as a part of the standard. Rust has its portable SIMD module but that too is in its early stages. Even if these efforts prevail and are adopted in their respective ecosystems, they still do not entirely solve the problem of portability as many instructions do not have direct equivalents across different ISAs. So, even though these efforts may prove useful in the average application, those concerned with maximum efficiency will still rely on using the intrinsics directly. Intrinsics aren't going anywhere and their difficult usage means most applications aren't going to make the effort to target multiple CPU architectures. At best, these applications will fall back to scalar code which cannot always be easily auto-vectorized by modern compilers.

The llsimd project aims to enable programs written using SIMD intrinsics to be able to run on any[1] CPU architecture. We do this by leveraging the target-independent nature of LLVM IR's first-class support of vector operations. Some instructions (e.g. pmaddwd) may need to be emulated as there is no equivalent LLVM IR representation, but our primary goal is portability. This allows preexisting projects to become more widely accessible and future-proof against the rise and decline of specific architectures over time.

Many projects have attempted to address to non-portable nature of SIMD by providing their own interfaces. Prominent examples include GCC's Vector Extensions, Clang's vector support and Google's Highway project. A major disadvantage of using these solutions, however, is that the user is "locked in" to that particular solution for the lifetime of your project unless they are willing and able to perform major refactoring efforts. A more direct equivalent to our project is the SIMD Everywhere (SIMDe) library. SIMDe is a header-only library which implements intrinsics in terms of target intrinsics. The hand-crafted nature of these implementations guarantees an optimal translation but requires a lot of effort. Adding a new target ISA, for example, would require manually writing out code for *all* ISAs already supported. This may prove unsustainable in the long term if the library wishes to continue supporting older intrinsics. Our project instead hands off the responsibility for the backend to LLVM which alleviates a lot of the burden of

[1] Instruction sets that have LLVM backends

development. Instead of having to write equivalent code for every target architecture, we only need to implement any new ISA in terms of LLVM IR and the rest is free. This also means our solution is future-proof as so long as LLVM supports a target backend, our project will also be able to support it. Since llsimd operates on LLVM IR directly, it can also theoretically[2] support languages other than C/C++ which have an LLVM frontend such as Swift, Rust, Zig and more.

Implementation Details

The llsimd project heavily relies on LLVM compiler infrastructure for transformation and compilation. The core of our project is an LLVM transformation pass which transforms IR calls to platform-specific SIMD instructions to generic vector operations.

Figure 1: Overview of the typical workflow using llsimd

Instruction Set

Due to time constraints and other reasons, we focused on implementing only MMX intrinsics for the purposes of the course project. We knew we only wanted to work with a single SIMD instruction set to start with and the first decision was to choose an ISA between x86 and ARM. Despite the recent growth in adoption of ARM driven primarily by Apple's transition to their M-series processors, x86 still holds a sizable lead in the desktop PC market. Although MMX is not particularly popular or widely used today, it is supported by all Intel and AMD processors that support later SIMD instruction sets for backward compatibility reasons. For those reasons along with MMX being a smaller and simpler instruction set, we decided to start with MMX. This choice is of little significance for the future of the project, however, as with the general framework of llsimd established, adding new instruction sets is only a matter of manual translation effort.

LLVM 20

LLVM 20.1.0 was released just before we began work on writing the IR transformations on May 4, 2025. Among other things, one of the changes—which was not mentioned in the release notes—was the conversion of many of the MMX intrinsics in terms of Clang's vector extensions. This inconspicuous commit among other things[3] greatly reduced the work we had to manually do. Almost all of the intrinsics that were trivially representable in LLVM IR were now already implemented as such. This left more complex—and interesting—instructions for us to implement which primarily comprised of pack and shift instructions.

---

[2] Not tested
[3] The primary objective was to abandon MMX instructions altogether in favour of SSE

Example

Given a C program with the following code:

```
__v2si result = _mm_madd_pi16(m1, m2);
```

The resulting LLVM IR as given by Clang would be:

```
%result = call <4 x i32> @llvm.x86.sse2.pmadd.wd(<8 x i16> %m1, <8 x
i16> %m2)
```

In this case, LLVM does not have a direct equivalent representation of a multiply-add instruction, so llsimd emulates its behaviour by replacing the above call with the following:

```
%sext1 = sext <8 x i16> %m1 to <8 x i32>
%sext2 = sext <8 x i16> %m2 to <8 x i32>
%mul = mul <8 x i32> %sext1, %sext2
%even = shufflevector <8 x i32> %mul, <8 x i32> undef, <4 x i32> <i32
0, i32 2, i32 4, i32 6>
%odd = shufflevector <8 x i32> %mul, <8 x i32> undef, <4 x i32> <i32 1,
i32 3, i32 5, i32 7>
%result = add <4 x i32> %even, %odd
```

Surprisingly, as noted in the Implementation Evaluations section, this does not appear to have a large performance impact despite LLVM not being able to translate this instruction back to a multiply-add.

Implementation Evaluations

We created a test suite with unit tests to ensure the correctness of our transformations. Although the test suite requires an x86 machine, we also ran each unit test manually on an ARM machine. We also ran benchmarks comparing the performance of the original and transformed programs to measure any performance hits as a result of the transformations.

Unit Tests

Being already deeply intertwined in the LLVM ecosystem, we chose to use the LLVM Integrated Tester (lit) to write our test suite. The goal of these tests was to ensure the correctness of our transformations so to make the addition of future tests and possibly dynamically generating inputs, it requires the intrinsics to be tested also be supported natively on the machine. We mostly only test intrinsics that we transform ourselves and not those already portable.

ARM

To confirm that the llsimd project actually does what it is intended to do, we needed to test on an architecture that does not natively support MMX instructions. We chose ARM as it is the most widely used non-x86 instruction set. For documenting reasons, we wanted to start with a clean slate for our test environment. For this, we used the open-source QEMU emulator to create an Ubuntu ARM image.

This is also when how we developed the cross-compilation workflow. C/C++ source code using MMX intrinsics typically include the mmintrin.h header file in which LLVM adds a define guard to ensure it is not used on non-x86 platforms as that would typically result in undefined behaviour. Rather than simply defining the header or supplying our own header file, we decided to use Clang's excellent cross-compilation facilities.

Benchmarks

We used the hyperfine[4] benchmarking tool to evaluate any performance loss between the original and transformed binaries compiled from our test suite.

Intrinsic	Original (µs)	Transformed (µs)	Change
`_mm_madd_pi16`	366.95	390.50	6%
`_mm_srl_si64`	381.42	391.27	3%
`_mm_mulhi_pi16`	394.69	404.41	2%
`_mm_slli_pi32`	367.97	374.58	2%
`_mm_slli_pi16`	370.01	376.06	2%
`_mm_sll_pi16`	388.54	394.05	1%
`_mm_slli_si64`	367.31	370.97	1%
`_mm_srli_pi16`	375.36	378.64	1%
`_mm_packs_pi16`	397.51	400.63	1%
`_mm_srai_pi16`	365.23	368.08	1%
`_mm_srai_pi32`	369.11	370.07	0%
`_mm_sll_si64`	374.75	372.29	-1%
`_mm_srl_pi32`	384.24	381.58	-1%

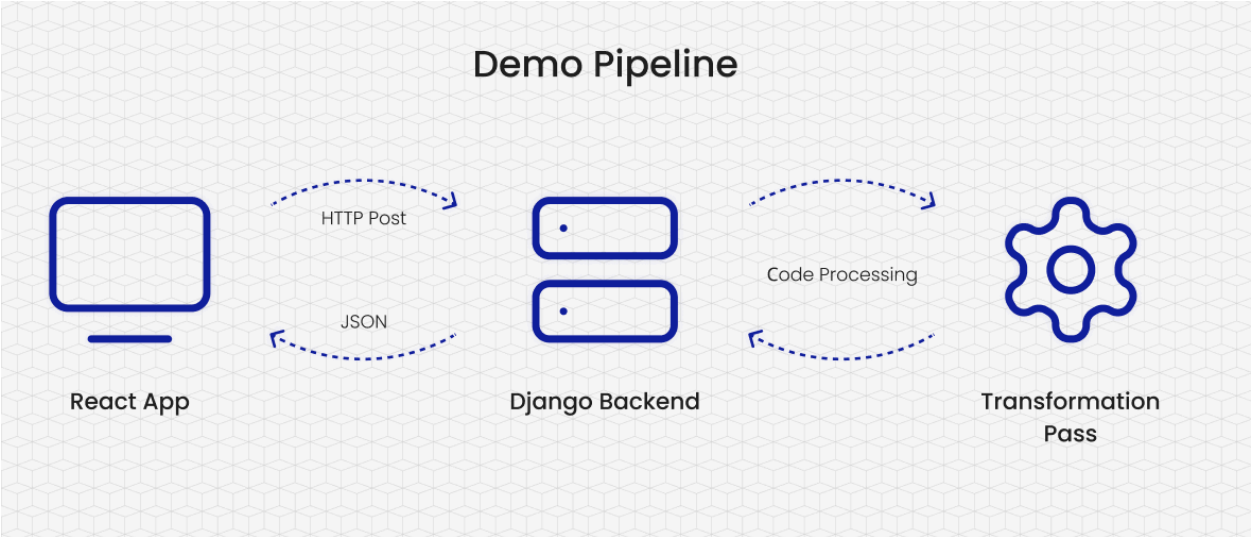
[4] perf did not support counting CPU cycles among other vitral information on our machines

Intrinsic	Original (µs)	Transformed (µs)	Change
`_mm_srli_pi32`	382.07	378.83	-1%
`_mm_packs_pu16`	391.15	387.32	-1%
`_mm_sll_pi32`	386.34	381.47	-1%
`_mm_srli_si64`	387.62	381.49	-2%
`_mm_srl_pi16`	365.65	359.00	-2%
`_mm_sra_pi16`	377.61	366.51	-3%
`_mm_sra_pi32`	371.56	358.62	-3%
`_mm_packs_pi32`	398.04	376.98	-5%

As expected, intrinsics such as `mm_madd_pi16` and `mm_mulhi_pi16` that require multiple instructions to emulate are among the slowest. We also notice that intrinsics like `mm_packs_pi32` are *faster* after our transformations than before. This could be a result of LLVM being able to use more modern instructions which have higher throughput.

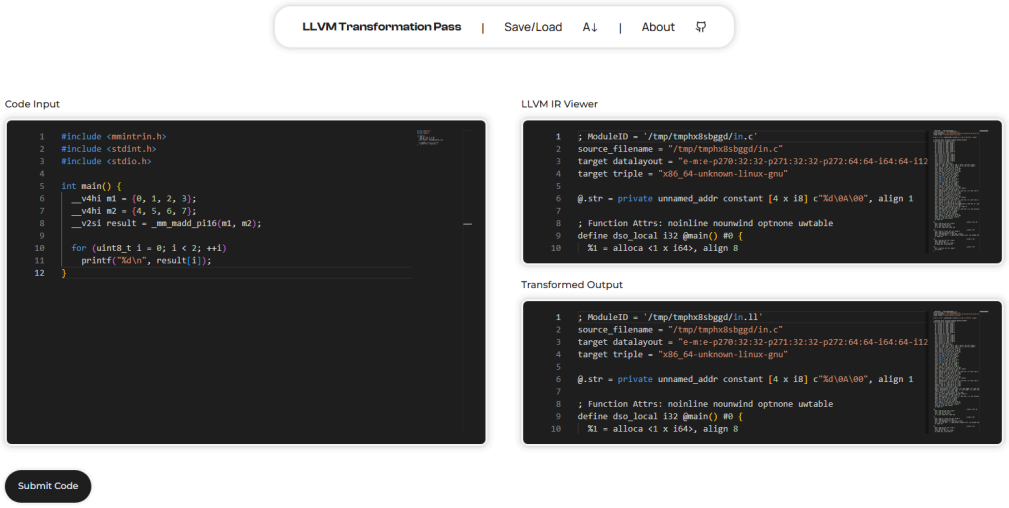
Demo

Our demo presents a complete end-to-end processing pipeline that integrates a React-based user interface with a Django back end to showcase real-time code transformation. The system demonstrates how user input can be sent from a modern web application to a high-performance back-end environment, where it is processed and then returned for display, all in a fully interactive experience.



The pipeline begins with a React application in which code is entered and submitted. That data

is conveyed as JSON to a Django backend, which receives and interprets the content before orchestrating further processing. The code is then passed to the transformation pass, which utilizes LLVM's capabilities to restructure or optimize it. This arrangement demonstrates a complete journey from a simple front-end submission to an advanced back-end transformation, illustrating how each segment in the chain cooperates to deliver enhanced performance and maintain portability.



The screenshot above captures the live demo, highlighting the interactive interface where code is input, and processed, and the results displayed.

Conclusion

The llsimd project successfully made the MMX instruction set portable across CPU architectures by leveraging LLVM IR. We demonstrated that not only do the transformed binaries have comparable performance, but sometimes even better than the original. More importantly, the llsimd project provides a framework on which further instruction sets can be easily implemented.

Lessons Learned

This project gave us an opportunity to learn about how SIMD intrinsics are used in code and what they do at a fundamental level. We also gained valuable experience working with the LLVM framework.

Future Work

The logical next step for the llsimd project is to expand to other instruction sets such as SSE, AVX and NEON. This would make the project more useful as these are more popular and modern.

As it is, multiple binaries have to be compiled for multiple target architectures. It would be convenient for the end-user if llsimd had first-class, built-in but optional support for dynamic dispatching.

If the project were to switch from its primary goal of portability to performance, the target architecture could be taken into account and the transformation pass could utilize target intrinsics that more closely match the input intrinsics than what LLVM IR offers.

For the project demo, better styling for the Monaco editor is planned to be implemented. This way our demo would be able to highlight the parts of the LLVM IR that were updated

Appendixes

Source Code

The main llsimd Git repository is available on GitHub:
https://github.com/bhavjitChauhan/llsimd

The live demo can be found at:
https://llsimd-demo.netlify.app

The demo Git repository is also available at:
https://github.com/antonkazachenko/llsimd-demo

Unit test shell commands

We run the following shell commands for each unit test file:

```
clang -S -emit-llvm %s -o %t.ll
opt -load-pass-plugin %root/build/libllsimd.so -passes=llsimd -S %t.ll
-o %t.out.ll
diff <(lli %t.ll) <(lli %t.out.ll)
```

QEMU Ubuntu ARM image creation

Ubuntu image from https://cloud-images.ubuntu.com/releases/oracular/release/.

Firmware image from
https://releases.linaro.org/components/kernel/uefi-linaro/16.02//release/qemu64/.

QEMU startup PowerShell script:

```
qemu-system-aarch64 `
      -M virt `
      -m 4096 `
      -cpu cortex-a710 `
      -smp 4 `
      -nographic `
      -bios .\QEMU_EFI.fd `
      -drive
if=none,file=.\ubuntu-24.10-server-cloudimg-arm64.img,id=hd0 `
      -device virtio-blk-device,drive=hd0 `
      -drive file=user-data.img,format=raw `
      -device virtio-net-device,netdev=net0 `
```

```
-netdev user,hostfwd=tcp:127.0.0.1:2222-:22,id=net0
```

Cross-compilation on ARM

```
clang --target=x86_64-unknown-linux-gnu -S -emit-llvm "$1" -o "$basename.ll"
opt -load-pass-plugin "$2" -passes=llsimd -S "$basename.ll" -o
"$basename.out.ll"
clang "$basename.out.ll" -o "$basename" &>/dev/null
```

Benchmark machine specifications

Ubuntu 24.10 on Windows 10 x86_64
CPU: 12th Gen Intel(R) Core(TM) i5-12600K
Memory: 15GiB

Benchmark script

```
for file in "$1"/*; do
      if [ -f "$file" ]; then
            basename=$(basename "$file" .c)
            .llsimd.sh "$file"
            clang "$basename.ll" -o "$basename.in"
            hyperfine  "$basename" "$basename.in" -N --warmup 10 --runs
100000 --export-csv "$basename.csv" --export-markdown "$basename.md"
--export-json "$basename.json"
      fi
done
```

Raw benchmark data

intrinsic	mean	stddev	median	min	max	out_mean	out_stddev	out_median	out_min	out_max
_mm_madd_pi16	366.9589	49.64015	354.7825	300.387	1430.468	390.5059	103.5219	364.6665	298.258	2023.907
_mm_mulhi_pi16	394.6903	114.101	363.8745	302.541	1893.618	404.414	112.6413	373.61	301.161	2807.251
_mm_packs_pi16	397.5155	113.2066	366.866	305.611	2058.025	400.6333	119.9359	369.6645	305.94	2772.46
_mm_packs_pi32	398.0439	112.4272	367.516	302.477	2063.399	376.9897	72.89455	358.614	302.189	1615.108
_mm_packs_pu16	391.1554	104.9742	365.4515	302.682	2365.999	387.3293	99.11388	363.1315	296.655	1963.247
_mm_sll_pi16	388.54	92.32584	366.067	303.49	1738.698	394.0545	92.87052	371.0915	305.225	3742.555
_mm_sll_pi32	386.3444	79.09411	365.8795	303.908	1848.412	381.4789	77.88622	360.873	302.547	4046.612
_mm_sll_si64	374.7537	77.78303	355.2565	301.871	1898.632	372.2918	65.2125	357.674	300.797	5540.097
_mm_slli_pi16	370.0152	61.60236	354.57	301.431	1868.848	376.0661	77.8369	357.276	302.183	2326.835

intrinsic	mean	stddev	median	min	max	out_mean	out_stddev	out_median	out_min	out_max
_mm_slli_pi32	367.9777	55.68307	354.807	305.718	1560.484	374.5811	64.17966	359.6755	305.215	3507.45
_mm_slli_si64	367.3181	55.13372	353.8275	304.201	1493.353	370.9744	60.9423	356.235	303.274	1676.663
_mm_sra_pi16	377.6153	70.97861	359.9645	303.752	4183.81	366.5174	54.61572	352.5745	299.989	1574.181
_mm_sra_pi32	371.5614	61.52746	355.3245	300.878	1539.134	358.6214	49.03293	345.997	300.764	1571.62
_mm_srai_pi16	365.2363	54.60009	351.355	298.476	1486.803	368.0804	55.51953	354.1785	292.498	1433.984
_mm_srai_pi32	369.1101	67.3111	351.916	296.722	3224.335	370.0798	68.24348	352.078	291.805	2076.615
_mm_srl_pi16	365.6573	55.25019	350.6055	293.758	1533.832	359.085	48.38122	346.2505	294.093	1409.566
_mm_srl_pi32	384.2422	91.65466	362.3165	296.048	1964.324	381.5824	83.05458	360.2025	301.505	2465.83
_mm_srl_si64	381.4245	87.01481	359.0515	303.962	2593.854	391.2733	89.89265	367.7135	305.624	2000.213
_mm_srli_pi16	375.3693	74.61838	356.5525	303.9	1737.733	378.6431	76.18208	359.256	299.992	2819.121

intrinsic	mean	stddev	median	min	max	out_mean	out_stddev	out_median	out_min	out_max
_mm_srli_pi32	382.0723	77.65723	362.4025	304.224	1725.871	378.8395	73.81745	359.133	302.493	1611.346
_mm_srli_si64	387.6212	87.18626	364.9165	305.713	1953.747	381.49	87.39002	359.008	302.756	2000.274