NEC SX-Aurora TSUBASA and the LLVM compiler infrastructure

Simon Moll[1], Matthias Kurtenacker[1], Erich Focht[2] and Sebastian Hack[1]

[1]Compiler Design Lab, Saarland University, Germany

SIC Saarland Informatics Campus

[2]NEC HPC Europe

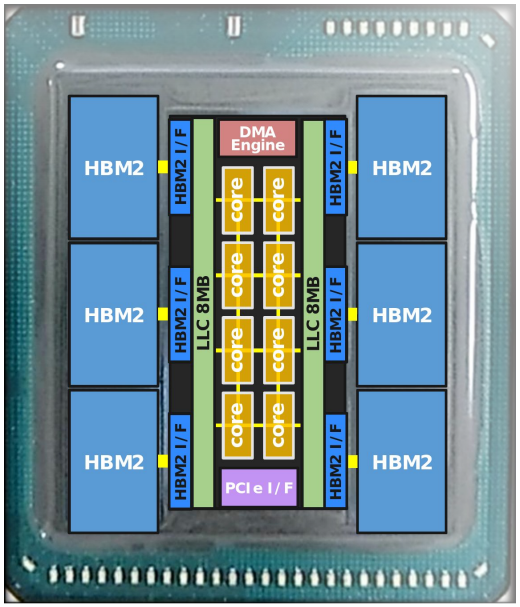
\Orchestrating a brighter world NEC

NEC SX-Aurora TSUBASA

High-performance vector CPU designed for sustained simulation performance.

NEC SX-Aurora TSUBASA

- 1.2 TB/s memory throughput
- up to 4.9 TFlops (f32)



LLVM

The leading infrastructure for static compilation today. Backed by..

- **NVIDIA** GPU driver stack.
- **ARM** Foundation of official ARM compiler.
- **AMD** GPU and AOCC, official compiler for AMD x86 CPU.
- **Intel** future replacement for Intel C Compiler, also GPU stack/NEO OpenCL.
- **Apple** Xcode compiler for the entire Apple ecosystem.
- **Google** Internal use for very large projects, official compiler for Android.
- **Qualcomm** Official Snapdragon LLVM compiler.
- **Xilinx** Vivado High-Level Synthesis compiler for FPGAs.
- ...

Why LLVM for NEC SX-Aurora TSUBASA?

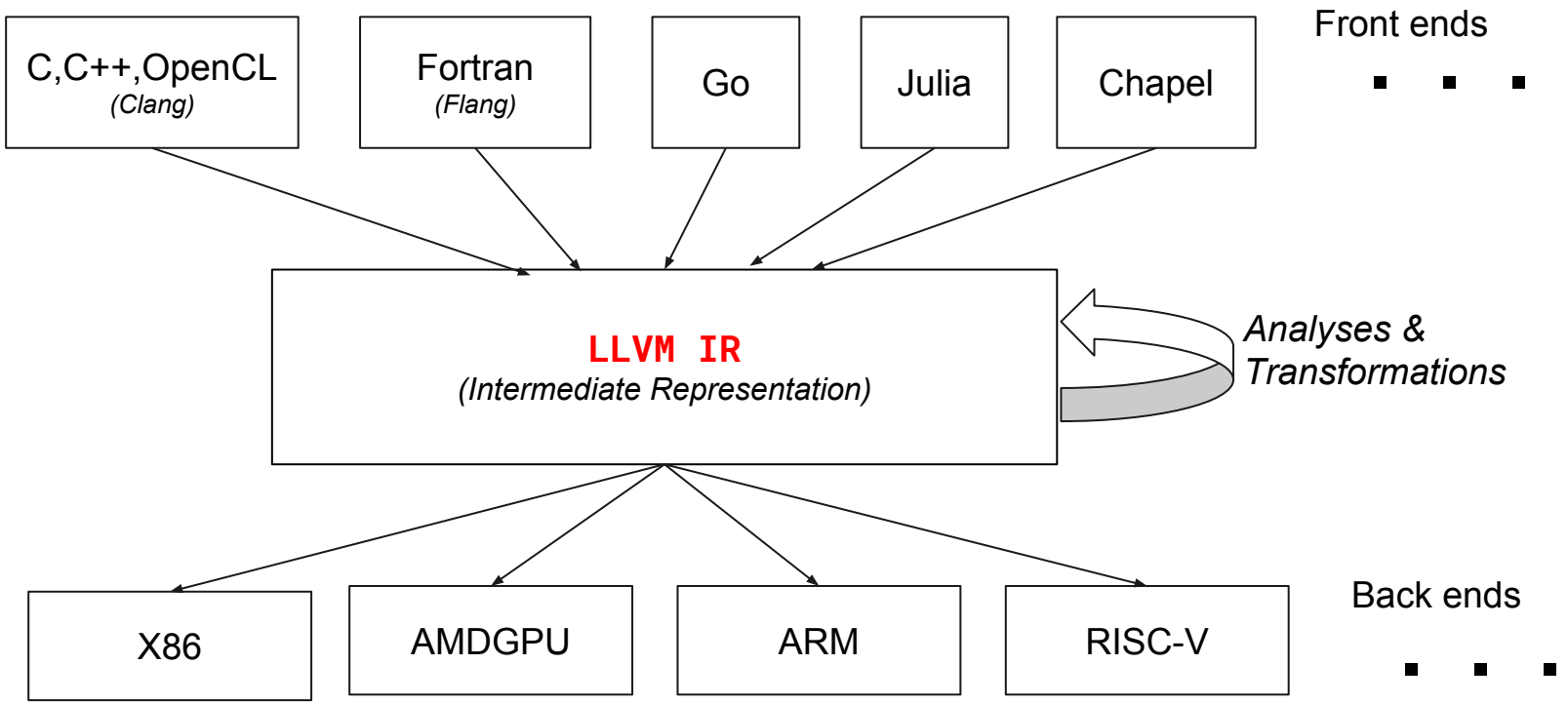
1. Production-grade compiler toolchain that is actively backed by industry.

2. An LLVM backend implies access to the LLVM umbrella.

3. Open-source project with a permissive license.

4. Great infrastructure for compiler research.

LLVM infrastructure



LLVM Umbrella

LLDB (replacement for gdb)

Polly (polyhedral optimization)

Clang

libopenmp
("#pragma omp parallel for" support in Clang)

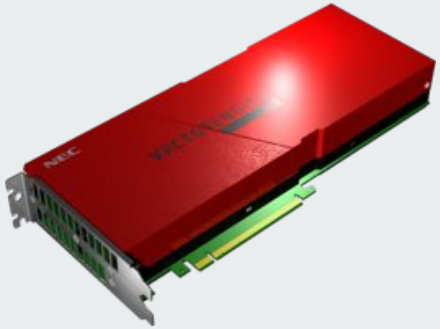
libcxx/libcxxabi
(standard-conformant, C++14)

POCL (OpenCL driver)

Plugins

Region Vectorizer

libYourTransformation.so



NEC SX-Aurora TSUBASA

LLVM

Is LLVM a good match? Consider..

- Frontend (Compared to NCC, the NEC C/C++ compiler)
- Optimization (Vectorization & Parallelization)
- Backend

Frontend: how does LLVM compare to NCC?

	C++	Fortran	OpenMP
LLVM	C++17 (complete, Clang)	2003, some 2008 (flang)	3.1 (full) / some 4,4.5
NCC	C++14 / C11	2003, some 2008	some 4

- **NCC** NEC C/C++ Compiler for SX-Aurora TSUBASA.
- Clang is command-line compatible with GCC (works well with cmake). NCC not so much.
- Lacking OpenMP 4 support in LLVM mostly due to vectorization..

Optimization: How does LLVM compare to NCC?

NCC's compiler has strong support for **automatic loop vectorization** and **parallelization.**

Loop Vectorization

- Weak spot in LLVM at the moment (only inner-most loop, no branches).
- Work is underway to improve loop and function vectorization in LLVM (Intel **VPlan**).
  - Progress is slow. CDL is cooperating with Intel/AMD on this.
- The ***Region Vectorizer*** enables outer-loop vectorization and function vectorization in LLVM.
  - OpenMP "`#pragma omp simd`" and "`#pragma omp declare simd`" in Clang.

Automatic Parallelization

- Support for "`#pragma omp parallel for`". No automatic parallelizer in LLVM.

RV - The Region Vectorizer

- Inter-procedural outer-loop and whole-function vectorizer (pragma driven).

```
#pragma omp simd safelen(8)
for (int i = 0; i < n; ++i)
  for (int j = 0; j < m; ++j)
    C[i] = pow(A[i], B[j]);
```

```
double pow(double x, double y) {
   […]
}

double8
pow_v8_vv(double8 x, double y) {
  [...]
}
```

auto-vectorized function

- Automatic recognition of conditional reduction patterns.

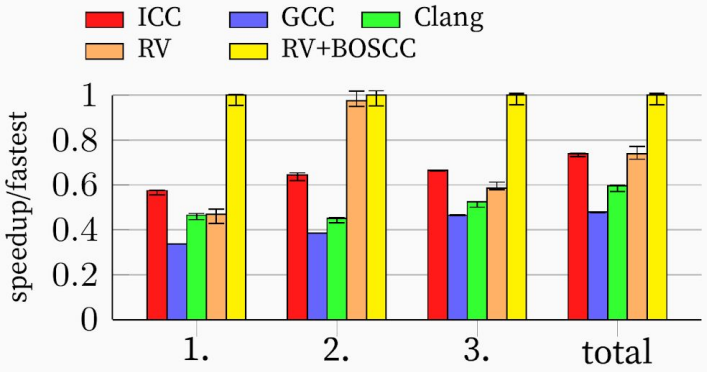
```
double a = 0.0;
for (int i=0..m) {
  if (A[i] > 42.0) a += B[i];
}
```

Available on github - https://github.com/cdl-saarland/rv

RV - PLDI'18 "Partial Control-Flow Linearization"

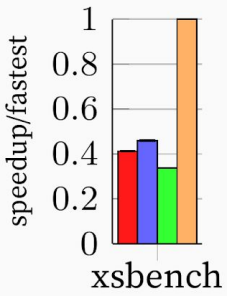
Case studies

SPEC2017 nab_s: BOSCC gadget



- Three hot loops (1, 2 & 3)
- RV+BOSCC: Three nested BOSCC-gadgets
  → unstructured control

XSBench: Divergent Loop Transform



Key kernel of the Monte Carlo neutronics application OpenMC

- Irregular data-structure traversal inside a vectorizable loop

RV's strength is control-flow preservation in code with mixed uniform and data-dependent branches.

LLVM IR support for SIMD

LLVM IR has built-in support for short SIMD types.

`<8 x double>` vector of 8 doubles

`<8 x i1>` predicate of 8 bits (for example on AVX2),

.. and predication (mask registers).

```
<8 x double> llvm.masked.load(double * basePtr, <8 x i1> predicate, <8 x double> defaultValue)
```

However, NEC SX-Aurora is wide SIMD with a ***parametric vector length!***

```
VL := 3  // ← active vector length register

X = vfadd [1,2,3,4,5], [1,2,3,4,5] // ← only computes three lanes.

X == [2,4,6,_,_]
```

This is unsupported in LLVM ... (yet).

LLVM Scalable Vector Extension

```
<scalable 2 x double> %v
```

%v is a `double` vector whose length is *some* multiple of two. The exact length is hardware dependent.

1. Originally proposed by ARM to support the ARM SVE vector ISAs.

   Why? CPUs will only scale in parallelism not in speed. The same ARM SVE binary will run twice fast if vector length is doubled on next iteration of hardware.

2. ARMs internal compiler already uses LLVM-SVE for vectorization.

3. RISC-V is looking into LLVM-SVE for the V extension.

4. SX-Aurora's ISA (VE) is vector-length parametric as well (close to RISC-V V extension).

=> Develop a VE backend for LLVM-SVE.

LLVM-VE: A LLVM Backend for SX-Aurora

Cooperation of NEC (Ishizaka-san, Marukawa-san, Erich Focht) and the Compiler Design Lab at Saarland University.

- **Objectives**: (1) develop a LLVM-SVE backend for NEC SX-Aurora. (2) Explore potential of advanced vectorization techniques.

- Poster & talk at the upcoming US LLVM Developers' Meeting to coordinate with LLVM community.

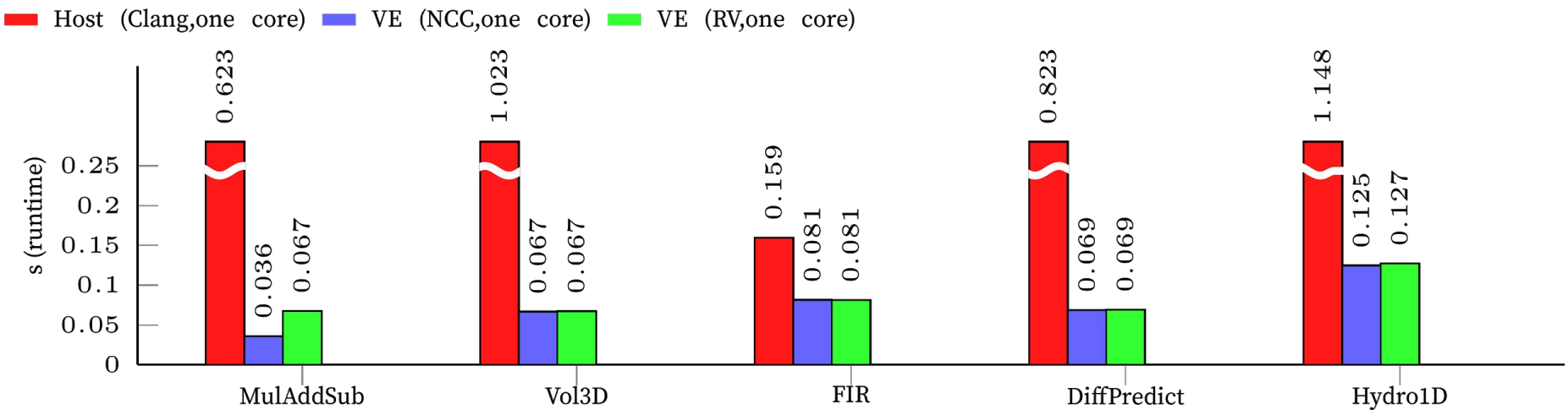
  -> NEC can influence the development of LLVM-SVE to better suit their hardware.

Early results

- Prototype backend for "classic" LLVM IR (<256 x double> to represent a VE vector register).

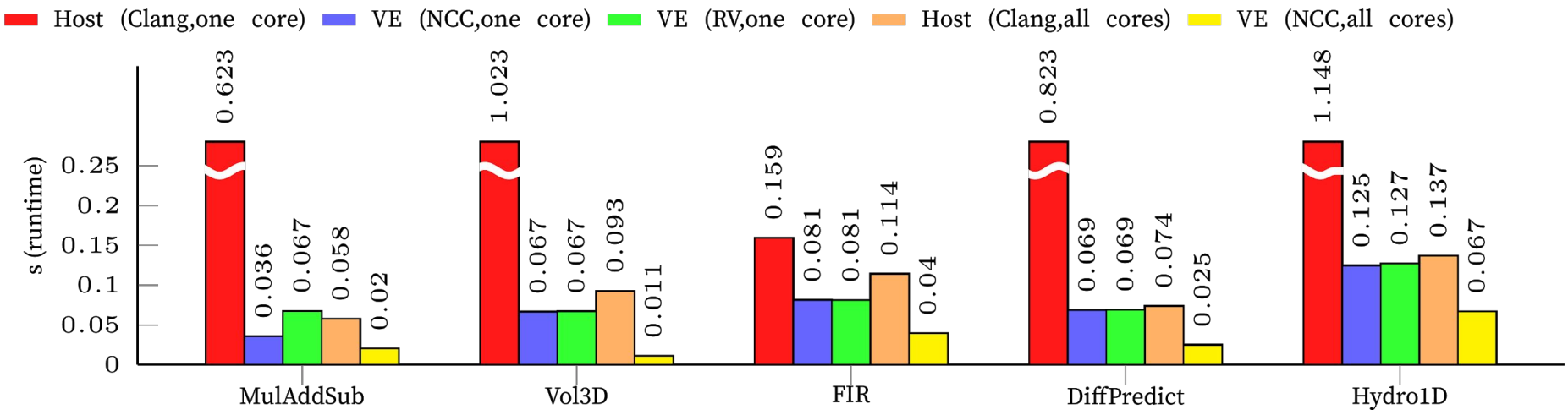
- RAJAPerf benchmark suite of LLNL. Kernels ported to C (from C++) for lack of libcxx support. Vectorized with the *Region Vectorizer (RV)*.

RAJAPerf (single core)



- LLVM-VE backend for classic vector IR (ie *<256 x double>*)
- LLVM-VE + Outer-loop vectorization with the Region Vectorizer (RV)
- Matched single core performance

RAJAPerf (all cores)



- Host: Skylake CPU (12 cores, HT, 2.6GHz) with AVX512 (using LLVM's LoopVectorizer).
- Results for OpenMP "`#pragma omp parallel for`"
- Performance potential for LLVM-VE.

Research topics - a selection

- Automatic vector compact/expand placement to improve utilization in conditional code.

```
for (int i=0..n) {
  v = B[i];
  if (foo(v)) {
    bar(v);
  }
}
```

```
for (int i=0..n, i += 256) {
  setvl(256)
  v = vload_256(B[i:256]);
  bool256 M = foo(v);
  x = compact_v256(v, M); ← tightly packs v where M is true
  setvl(popcount(M)); ← shorten VL
  bar(x);
}
```

- Multi-dimensional vectorization (RV prototype). Exploit 2D memory accesses on the SX-Aurora.
- Cost modeling (compiler/vectorizer heuristics).
- New applications (e.g. raytracing, databases, ..).

Conclusion

- Now is the right time to develop a NEC SX-Aurora backend for LLVM.
  a. The LLVM community is finalizing the design of LLVM-SVE.
  b. Advanced vectorization capabilities are being developed for LLVM (*VPlan* upstream, *RV* as a plugin).

- Initial results show promising results.
  a. LLVM-VE matches the performance of NCC for single core.
  b. RAJAperf all cores results show potential for parallelism.

- #TODO
  a. LLVM-SVE backend.
  b. libOpenMP for SX-Aurora.
  c. Glibc port to compile libcxx/libcxxabi for SX-Aurora.

Backup Slides

RV for LLVM-VE

- LLVM has no notion of a Vector Length Register.

- Modified RV to control the *Vector Length register* VL (instead of remainder loop).

```
for (int i = 0, i < n; i += 256) {
  llvm.ve.lvl(min(n - i, 256)) // ← builtin function to set VL
  <vectorized body>
 }
```

- Enables outer-loop vectorization for NEC SX-Aurora with a native LLVM+RV code path.

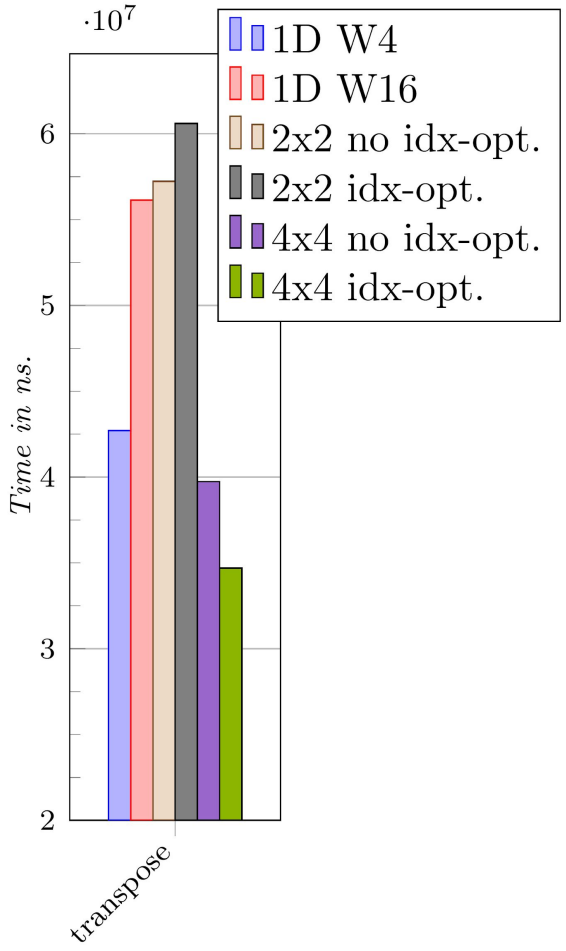
RV: multi-dimensional vectorization

```
transpose(
   double * A, double * B, int n) {

for (int i = 0; i < n; ++i) {
  for (int j = 0; j < n; ++j) {
    B[i*n+j] = A[j*n+i]
  }
}

}
```

```
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < n; ++j) {
    V0 = vload_v4(A[i][j])
    V1 = vload_v4(A[i+1][j])
    V2 = vload_v4(A[i+2][j])
    V3 = vload_v4(A[i+3][j])
    S0=shuffle(V0[0], V1[0]. V2[0], V3[0])
    S1=shuffle(V0[1], V1[1]. V2[1], V3[1])
    S2=shuffle(V0[2], V1[2]. V2[2], V3[2])
    S3=shuffle(V0[3], V1[3]. V2[3], V3[3])
    vstore_v4(B[i][j], S0)
    vstore_v4(B[i+1][j], S1)
    vstore_v4(B[i+2][j], S2)
    vstore_v4(B[i+3][j], S3)
  }
}
```



Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz (AVX2)

RV: multi-dimensional vectorization

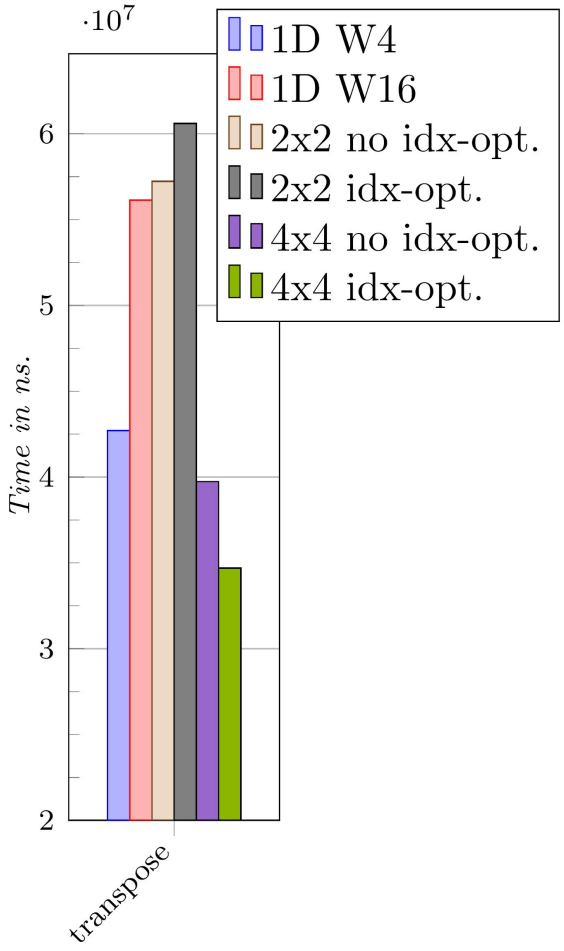
```
transpose(
   double * A, double * B, int n) {

for (int i = 0; i < n; ++i) {
  for (int j = 0; j < n; ++j) {
    B[i*n+j] = A[j*n+i]
  }
}

}
```

```
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < n; ++j) {
    V0 = vload_v4(A[i][j])
    V1 = vload_v4(A[i+1][j])
    V2 = vload_v4(A[i+2][j])
    V3 = vload_v4(A[i+3][j])
    S0=shuffle(V0[0], V1[0]. V2[0], V3[0])
    S1=shuffle(V0[1], V1[1]. V2[1], V3[1])
    S2=shuffle(V0[2], V1[2]. V2[2], V3[2])
    S3=shuffle(V0[3], V1[3]. V2[3], V3[3])
    vstore_v4(B[i][j], S0)
    vstore_v4(B[i+1][j], S1)
    vstore_v4(B[i+2][j], S2)
    vstore_v4(B[i+3][j], S3)
  }
}
```

```
vload_4x4(A[i][j])
vstore_4x4(B[j][i])
```



Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz (AVX2)

Divergence Analysis (LLVM patch)

https://reviews.llvm.org/D50433