

Using CLANG/LLVM Vectorization to Generate Mixed Precision Source Code

Régis PORTALEZ — ALTIMESH — regis.portalez@altimesh.com — Florent DUGUET — ALTIMESH — florent.duguet@altimesh.com

MIXED PRECISION DEVICES

At Supercomputing 2015, NVIDIA announced Jetson TX1, a mobile supercomputer, offering up to 1 TFLOPs of compute power for a power envelope typical of embedded devices. Targeting image processing and deep learning, this platform is the first available to natively expose mixed precision instructions. However, the new mixed precision unit requires that operations on 16-bit precision floating points are done in pairs. Hence, approaching peak performance level requires usage of the half2 type which pairs two values in a single register.

In this work, we present an approach that makes use of existing vectorization tool developed for CPU code optimization to further generate CUDA source code that uses half2 intrinsic functions, hence enabling mixed precision hardware usage with little effort. Using this approach, we are able to generate efficient CUDA code from a single scalar version of the code.

This source to source code translation may be used in many application fields for different numeric types. Moreover, this approach shows very nice boundary effects such as better memory access pattern and instruction level parallelism.

APPROACH

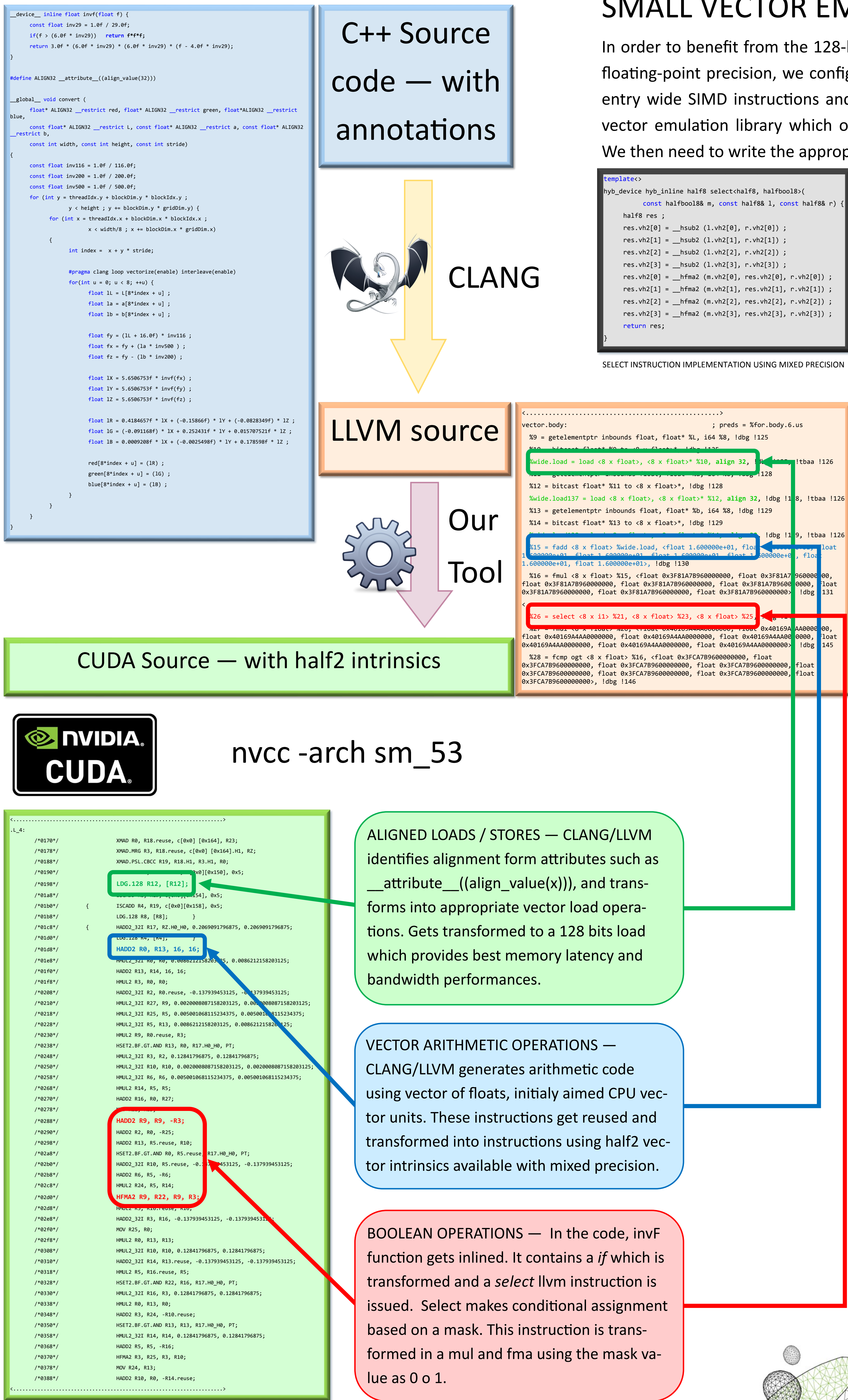
The Jetson TX1 and the upcoming Pascal are capable of **mixed precision**. This feature aimed at deep learning and image processing offers a 16-bit floating-point arithmetic unit with improved performances. The way this unit is exposed [1] is similar to SSE instructions, that is each operation is performed on a pair of values. SSE instructions, and more generally SIMD instructions have been around for more than a decade.

Compilers and optimizers are familiar with these instruction sets and generate code that makes use of those. The LLVM compiler [2] is an example of choice: indeed, its intermediate representation performs the vectorization stage operating on small vector registers [3]. Then, these vectors are translated into small vector instructions by the backend, depending on the target platform — ARM with NEON, Intel x86 with SSE or AVX, IBM Power with VSX.

Building on the analogy between mixed precision instructions and SIMD instructions, we make use of the vectorization unit of CLANG operating on C++ together with LLVM to generate intermediate language with small vectors. However, a few modifications have to be made:

- 16-bits floating-point precision (a.k.a. half) is not supported on most SIMD units, hence, the input source code is written using 32-bit precision (float). In the transformation from LLVM intermediate to CUDA, we convert float32 into float16 automatically.
- The SIMD unit used by LLVM is aligned with known architecture sizes which are at least 128 bits (SSE, NEON, VSX). However, mixed precision is 32 bits with two halves. The transformation is not a one-to-one: a kind of emulation library has to be written.
- Some operations assumed to exist in the SIMD instruction set of a GPU do not exist in mixed precision, such as *select* [4] (x86 : *vblend*, Altivec/*VSX*: *vec_sel*). These operations have to be written in the emulation library.

As illustrated here, 128-bit aligned memory access provide best bandwidth. Hence, we use the 8-entries wide small vector registers from LLVM, as 8 halves have a 128-bit memory footprint.



SMALL VECTOR EMULATION LIBRARY

In order to benefit from the 128-bit aligned loads, even using 16-bit floating-point precision, we configure the LLVM optimizer to use 8-entry wide SIMD instructions and registers. Then, we write a small vector emulation library which operates on a custom type **half8**. We then need to write the appropriate operators on these entries.

```
template<
__device__ inline half8 select(half8, half8) {
    const half8 m, const half8 r, const half8 r {
        half8 res;
        res.vh2[0] = __hsub2 (l.vh2[0], r.vh2[0]);
        res.vh2[1] = __hsub2 (l.vh2[1], r.vh2[1]);
        res.vh2[2] = __hsub2 (l.vh2[2], r.vh2[2]);
        res.vh2[3] = __hsub2 (l.vh2[3], r.vh2[3]);
        res.vh2[0] = __hfmaz (m.vh2[0], res.vh2[0], r.vh2[0]);
        res.vh2[1] = __hfmaz (m.vh2[1], res.vh2[1], r.vh2[1]);
        res.vh2[2] = __hfmaz (m.vh2[2], res.vh2[2], r.vh2[2]);
        res.vh2[3] = __hfmaz (m.vh2[3], res.vh2[3], r.vh2[3]);
        return res;
    }
}
```

SELECT INSTRUCTION IMPLEMENTATION USING MIXED PRECISION

We illustrate here the details of the select operation, most others being quite straightforward.

APPLICATION TO IMAGE PROCESSING

When operating on images, some algorithms operate on different color spaces. We illustrate here the conversion between CIE-L*a*b* [5] to RGB color-space, via the CIE-XYZ color space. Note that the transformation calls a function that contains an *if* clause, which results in a *select* in LLVM intermediate language.

Other image processing algorithms such as filters could be applied, however, the benefit of this approach is limited to the vectorization capability of the LLVM optimizer.

DISCUSSION

This approach reuses major contributions to the compilers and optimizer research fields. Indeed, the data alignment and automatic vectorization including transformation of conditions into select instructions are very hard to identify and implement. The implementation of small vector library and float to half transformation is pretty straightforward and does not show major technical lock.

While this work is motivated by the mixed-precision intrinsics library, and its SIMD nature, it can also be used without the final float to half transformation. Indeed, on Kepler for instance, more execution units than scheduling units are present, which requires the scheduler to issue two fp32 operations in a single clock cycle to benefit from the full compute power. As a result, using a small vector library with several floats is a good way to offer the scheduler opportunities for ILP. Another significant improvement is the use of aligned 128-bit loads. Using the same input code compiled with CUDA, on a Maxwell architecture leads to bandwidth improvement (as illustrated above). When using 32-bit loads, the global read buffer gets saturated while reaching only 46% of memory bandwidth. A higher bandwidth usage is immediately obtained, without code change, automatically transforming 32-bit loads into 128-bit loads.

(In this example, the loop is unrolled and both kernels are run with the same configuration and require the same number of registers).

These performance improvements may come at some supplemental cost: operating on four or eight values at a time may lead to more register pressure. A heuristic needs to be found to identify whether the benefit of using 128-bit loads is not lost with register spilling. Further improving the float to half transformation tool including a vector-size transformation might actually help in this operation. However, this would require significant code analysis as loop iterations count would be different.

REFERENCES

- [1] Mixed Precision — http://docs.nvidia.com/cuda/cuda-math-api/group_CUDA_MATH_HALF2_ARITHMETIC.html#group_CUDA_MATH_HALF2_ARITHMETIC
- [2] LLVM Compiler infrastructure: <http://llvm.org/>
- [3] "Vector LLVA: A Virtual Vector Instruction Set for Media Processing", Robert L. Bocchino Jr. and Vikram S. Adve. Proceedings of the Second International Conference on Virtual Execution Environments (VEE '06), Ottawa, Canada, 2006.
- [4] LLVM select instruction : <http://llvm.org/docs/LangRef.html#select>
- [5] L*a*b* Color Space from CIE: https://en.wikipedia.org/wiki/Lab_color_space

Régis PORTALEZ — ALTIMESH — regis.portalez@altimesh.com
Florent DUGUET — ALTIMESH — florent.duguet@altimesh.com