Document Number: N4454
Date: 2015-04-10
Project: Programming Language C++, Library Working Group
Reply-to: Matthias Kretz <kretz@compeng.uni-frankfurt.de>

SIMD Types Example: Matrix Multiplication

ABSTRACT

This document describes one possible implementation of a matrix class and matrix multiplication using the data-parallel SIMD types introduced in [N4184]. The example shows the basic use of SIMD types for manual transformation of a loop over scalars to a loop with increased stride using SIMD vector loads and stores and SIMD operations in the loop body.

CONTENTS

0 Remarks 1
1 Matrix Multiplication Introduction 1
2 Matrix Memory Layout 1
3 Matrix Multiplication 2
4 Discussion 7
A Acknowledgements 9
B Bibliography 9

0 Remarks

- In the following $\mathcal{W}_{\texttt{T}}$ denotes the number of scalar values (width) in a SIMD vector of type `T` (sometimes also called the number of SIMD lanes)
- Matrix multiplication is not the best motivating example for the unique features of SIMD types. The reason this example is important is because matrix multiplication is a relatively simple algorithm (though hard to implement with maximum efficiency) and a well-known and well-researched problem.

1 Matrix multiplication introduction

Matrix multiplication takes two two-dimensional arrays as input (sizes $N \times K$ and $K \times M$) and produces one two-dimensional array as output (size $N \times M$). Equation (1) shows an example with $(4 \times 3) \cdot (3 \times 2) \to (4 \times 2)$. The values for matrix C are calculated as $c_{i,j} = \sum_k a_{i,k} b_{k,j}$.

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \\ a_{3,0} & a_{3,1} & a_{3,2} \end{bmatrix} \cdot \begin{bmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \\ b_{2,0} & b_{2,1} \end{bmatrix} = \begin{bmatrix} c_{0,0} & c_{0,1} \\ c_{1,0} & c_{1,1} \\ c_{2,0} & c_{2,1} \\ c_{3,0} & c_{3,1} \end{bmatrix} \tag{1}$$

A C++ class for arbitrarily sized matrices is not overly hard to implement, but it distracts from the objective of this document. Therefore, in the following I will only discuss square matrices (i.e. $(N \times N) \cdot (N \times N) \to (N \times N)$).

2 Matrix memory layout

There are two conventional memory layouts used for storing matrices:

row-major Consecutive elements of the *rows* of the matrix are contiguous in memory. Thus the matrix B in (1) is stored as $[b_{0,0}, b_{0,1}, b_{1,0}, b_{1,1}, ...]$. Consequently, row-vectors out of the matrix are contiguous in memory.

column-major Consecutive elements of the *columns* of the matrix are contiguous in memory. Thus the matrix B in (1) is stored as $[b_{0,0}, b_{1,0}, b_{2,0}, b_{0,1}, ...]$. Consequently, column-vectors out of the matrix are contiguous in memory.

```
for (size_t i = 0; i < N; ++i) {
  for (size_t j = 0; j < N; ++j) {
    C[i][j] = A[i][0] * B[0][j];
    for (size_t k = 1; k < N; ++k) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Listing 1: The basic matrix multiplication algorithm.

Other alternatives, such as recursive storage of sub-matrices are also used. To simplify the discussion in this document I will only consider the "native" layout of C/C++. An array declared as `float A[N][N]` and accessed as `A[i][j]` is contiguous in memory in the `j` index and therefore uses *row-major* storage.

3 MATRIX MULTIPLICATION

Given three arrays declared as `float A[N][N]`, `float B[N][N]` and `float C[N][N]`, the matrix multiplication $A \cdot B = C$ can be implemented as shown in Listing 1. For large matrices this algorithm is cache-inefficient. The cache efficiency can be improved via blocking. The resulting implementation may use up to 12 nested loops instead of the three shown in Listing 1.

For large matrices, cache optimization is ultimately more important than vectorization of the algorithm. Since the purpose of this document is to show an example use of [N4184] with a well-known problem, I will leave cache-optimization as exercise for the reader and focus only on vectorization. In the following we therefore consider the caches to be large enough for our matrix sizes.

3.1 VECTORIZATION OPPORTUNITIES

The algorithm executes N^3 multiplications, all of which are independent and could execute in any order. The algorithm also executes $(N-1)N^2$ additions, all of which depend on the result of either a multiplication or another addition. Floating-point addition is not commutative. However, for matrix multiplication we may reorder the additions because it only distributes the statistical error differently.

Note that the code in Listing 1 lost all this information when the mathematical rules for matrix multiplication were transferred to a C++ algorithm. Of course, the compiler may transform the loop under the "as-if" rule, but ultimately has to reconstruct information the developer was unable to express in the algorithm. Explicit vectorization

```
1  using V = Vector<T>;
2  for (size_t i = 0; i < N; ++i) {
3    for (size_t j = 0; j < N; j += V::size()) {
4      V c_ij = A[i][0] * V(&B[0][j], Aligned);
5      for (size_t k = 1; k < N; ++k) {
6        c_ij += A[i][k] * V(&B[k][j], Aligned);
7      }
8      c_ij.store(&C[i][j], Aligned);
9    }
10 }
```

Listing 2: Basic vectorization of the matrix multiplication algorithm.

(partially) solves this issue of lost information about data-parallelism. Explicit vectorization can either be expressed via loops with vector semantics (e.g. [N3831]) or types with data-parallel operations. In the following I will discuss the latter approach.

One approach for vectorization of matrix multiplication builds upon vectorizing scalar products (row-vectors times column-vectors). (e.g. $c_{0,0} = a_{0,0}b_{0,0} + a_{0,1}b_{1,0} + a_{0,2}b_{2,0}$) Since this requires a reduction and leads to a scalar store, this is not the most efficient vectorization.

A second approach multiplies a column-vector from A with a single scalar entry from B which is broadcast to a full SIMD vector:

$$\left[c_{i,j}^{(k)}, c_{i+1,j}^{(k)}, c_{i+2,j}^{(k)}, ...\right] = \left[a_{i,k} \cdot b_{k,j}, a_{i+1,k} \cdot b_{k,j}, a_{i+2,k} \cdot b_{k,j}, ...\right]$$

The resulting $c_{i,j}^{(k)}$ values subsequently need to be added up: $c_{i,j} = \sum_k c_{i,j}^{(k)}$. These summations can execute independently for different (i, j) pairs. Thus, we reuse the $\left[c_{i,j}^{(k)}, c_{i+1,j}^{(k)}, c_{i+2,j}^{(k)}, ...\right]$ vectors to execute the additions in parallel. The resulting vector in C is again a column-vector.

A third approach uses row-vectors instead of column-vectors, but otherwise the same idea as above:

$$\left[c_{i,j}, c_{i,j+1}, c_{i,j+2}, ...\right] = \sum_k \left[a_{i,k} \cdot b_{k,j}, a_{i,k} \cdot b_{k,j+1}, a_{i,k} \cdot b_{k,j+2}, ...\right] \tag{2}$$

Since we will use row-major storage for the matrices, this leads to efficient SIMD vector loads and stores and thus the preferred vectorization approach.

The scalar algorithm from Listing 1 can thus be rewritten with the SIMD types from [N4184] as shown in Listing 2. Lines 4–6 calculate $\mathcal{W}_\mathrm{T}$ entries for the result matrix C in parallel. In line 4, one entry from A is read at $a_{i,0}$ and $\mathcal{W}_\mathrm{T}$ entries from B are read in parallel at $\left[b_{0,j}, b_{0,j+1}, b_{0,j+2}, \dots, b_{0,j+\mathcal{W}_\mathrm{T}-1}\right]$. The multiplication between these two objects expresses a component-wise multiplication of two SIMD vectors. There-

```
1  template <typename T, size_t N>
2  class Matrix {
3    using V = Vector<T>;
4    static constexpr size_t NPadded = (N + V::size() - 1) / V::size() * V::size();
5    using RowArray = std::array<T, NPadded>;
6    alignas(V::MemoryAlignment) std::array<RowArray, N> data;
7
8   public:
9    RowArray &operator[](size_t i) { return data[i]; }
10   const RowArray &operator[](size_t i) const { return data[i]; }
11 };
```

Listing 3: The declaration of a matrix class.

fore, the $a_{i,0}$ value is broadcast to a full SIMD vector with $\mathcal{W}_\mathtt{T}$ entries, all containing the $a_{i,0}$ value. The variable `c_ij` thus stores the result of

$$\left[a_{i,0} \cdot b_{0,j},\ a_{i,0} \cdot b_{0,j+1},\ a_{i,0} \cdot b_{0,j+2},\ \ldots,\ a_{i,0} \cdot b_{0,j+\mathcal{W}_\mathtt{T}-1}\right]$$

. In line 6 the results of

$$\left[a_{i,k} \cdot b_{k,j},\ a_{i,k} \cdot b_{k,j+1},\ a_{i,k} \cdot b_{k,j+2},\ \ldots,\ a_{i,k} \cdot b_{k,j+\mathcal{W}_\mathtt{T}-1}\right]\ \forall 0 < k < N$$

are added component-wise to `c_ij`. The algorithm thus implements the parallel expression of the matrix multiplication formulated in equation 2. Line 8 finally stores the $\mathcal{W}_\mathtt{T}$ scalar values from the local variable `c_ij` to the two-dimensional array `C`, starting at the address `&C[i][j]` and overwriting $\mathcal{W}_\mathtt{T}$ consecutive values of type `T`.

The implementation in Listing 2 makes two assumptions about the `B` and `C` arrays:

1. The allocated row size of the matrix storage is a multiple of `V::size()` (line 3) and thus larger than `N`, if necessary.
2. The addresses of the first entries (`&X[0][0]`) are aligned correctly for aligned loads and stores (lines 4, 6, and 8).

Consequently, a `Matrix<T, N>` class can be declared as shown in Listing 3.

3.2 vector width

The matrix multiplication algorithm and the `Matrix` class as shown in Listings 2 and 3 are portable to different targets with different $\mathcal{W}_T$. Thus, a target without SIMD support could use $\mathcal{W}_T = 1$ and an accelerator card a large value, such as $\mathcal{W}_T = 32$ or larger. In an abstract view, the description of the matrix multiplication algorithm in Listing 2 describes an arbitrary width of data-parallelism.

Note, however, that for large $\mathcal{W}_T$ the memory overhead for small matrices may be unreasonably large (because of the row padding for alignment). For small matrices the `Vector<T>` class might not be the best solution for fully portable and highly efficient code. `SimdArray<T, N>`[1] might be better suited for small matrices.

3.3 optimizations

Besides caches for large matrices, the load/store unit is a limiting factor already for small matrices. This can be optimized via unrolling in such a way that fewer loads must be executed. By unrolling the outer loop (over `i`) by a factor u, the vector load from `B[k][j]` can be reused u times. Listing 4 shows the complete code for a matrix multiplication. Note that this approach is valid for non-SIMD code as well. However, this example shows that the approach can be directly translated to the SIMD type variant. Note also that the inner loops execute the expression `V(&b[k][j], Vc::Aligned)` with the same `k` and `j`. The compiler has enough information available to optimize the multiple load expressions to a single load and reuses the register.

The second optimization in Listing 4 interleaves the final multiply-add with the store. The lines 20 and 36 could execute in the preceding loop. By moving them between the stores, the pressure on the store port is relieved.

If the code is compiled for a target system supporting FMA[2] instructions, the compiler can fuse the multiplication and addition operators in the middle- or back-end. This does not require extra effort in the `Vector<T>` implementation (e.g. expression templates could be used to generate FMA calls already in the library). The important point is that `Vector<T>` provides an intuitive syntax/API without loss of flexibility in the compiler middle- and back-ends.

1 I can describe this class in another paper if there is interest.
2 fused multiply-add

```
template <typename T, size_t N>
Matrix<T, N> operator*(const Matrix<T, N> &a, const Matrix<T, N> &b)
{
  constexpr size_t UnrollOuterloop = 4;
  using V = Vc::Vector<T>;
  Matrix<T, N> c;
  constexpr size_t i0 = N / UnrollOuterloop * UnrollOuterloop;
  for (size_t i = 0; i < i0; i += UnrollOuterloop) {
    for (size_t j = 0; j < N; j += V::size()) {
      V c_ij[UnrollOuterloop];
      for (size_t n = 0; n < UnrollOuterloop; ++n) {
        c_ij[n] = a[i + n][0] * V(&b[0][j], Vc::Aligned);
      }
      for (size_t k = 1; k < N - 1; ++k) {
        for (size_t n = 0; n < UnrollOuterloop; ++n) {
          c_ij[n] += a[i + n][k] * V(&b[k][j], Vc::Aligned);
        }
      }
      for (size_t n = 0; n < UnrollOuterloop; ++n) {
        c_ij[n] += a[i + n][N - 1] * V(&b[N - 1][j], Vc::Aligned);
        c_ij[n].store(&c[i + n][j], Vc::Aligned);
      }
    }
  }
  for (size_t j = 0; j < N; j += V::size()) {
    V c_ij[UnrollOuterloop];
    for (size_t n = i0; n < N; ++n) {
      c_ij[n - i0] = a[n][0] * V(&b[0][j], Vc::Aligned);
    }
    for (size_t k = 1; k < N - 1; ++k) {
      for (size_t n = i0; n < N; ++n) {
        c_ij[n - i0] += a[n][k] * V(&b[k][j], Vc::Aligned);
      }
    }
    for (size_t n = i0; n < N; ++n) {
      c_ij[n - i0] += a[n][N - 1] * V(&b[N - 1][j], Vc::Aligned);
      c_ij[n - i0].store(&c[n][j], Vc::Aligned);
    }
  }
  return c;
}
```

Listing 4: Matrix multiplication blocked on the outer loop for improved register reusage.

4 DISCUSSION

Matrix multiplication is a well-researched problem for current CPUs (to a large part because of LINPACK and the TOP500). Compilers are able to automatically vectorize some matrix multiplications and may completely fail on others (mainly for odd sizes). The unrolling/blocking optimizations are often the most important optimizations, though, and cannot be simplified through a data-parallel programming abstraction. For a developer that wants to optimize matrix multiplication for a given system it often is helpful to have full control over the vectorization and iteration approach. While only assembly language gives the highest control and can eliminate compiler differences, on the C/C++ level only intrinsics or SIMD types provide the necessary control.

4.1 AUTOMATIC VS. MANUAL OPTIMIZATION

Compilers are, in principle, able to reorganize loops and implement blocking for a given cache hierarchy. For applications depending on maximum efficiency for acceptable computing costs, it is often not feasible to rely on the compiler, though. Instead most of these projects rely on libraries that provide optimized (i.e. hand-tuned) implementations of such operations.

SIMD types can be used to implement such libraries. They provide the interface to access low-level optimizations with a standard C++ interface and at the same time reduce differences between different targets.

4.2 FIXED VECTOR WIDTH OR NOT?

In Listings 2–4, the vector width $\mathcal{W}_\mathrm{T}$ appears as `Vector<T>::size()`. This size is a constant expression and thus at compile time the SIMD types use a fixed width. The width used at compile time thus turns the matrix multiplication (i.e. the matrix rows) into chunks that work most efficient on the target hardware. This means that the objects used in the matrix multiplication (i.e. of type `Vector<T>`) abstract the data-parallel execution units of the target machine.

If the width were not fixed in terms of the type, the expression of the matrix multiplication would probably look like shown in Listing 5. Note that you can, except for the `store` function, use `std::valarray<T>` to actually implement Listing 5. The `store` line only needs to be replaced with `std::copy(std::begin(c_i), std::end(c_i), &C[i][0])`. Results of a benchmark compiled with GCC 4.9.1 is presented in Table 1.

```
using V = HypotheticalDataParallelType<T>;
for (size_t i = 0; i < N; ++i) {
  V c_i = A[i][0] * V(&B[0][0], N);
  for (size_t k = 1; k < N; ++k) {
    c_i += A[i][k] * V(&B[k][0], N);
  }
  c_i.store(&C[i][0]);
}
```

Listing 5: Matrix multiplication implementation using a hypothetical data-parallel vector type that does not have a fixed width at compile time.

`float` (auto-vectorized)	4.92	FLOP/cycle
`Vc::Vector<float>` (Listing 4)	9.67	FLOP/cycle
`std::valarray<float>` (Listing 5)	0.448	FLOP/cycle
`std::valarray<float>` (zero-copy)	1.67	FLOP/cycle

Table 1: Benchmark Results.

The issue with `valarray<T>` is that it is runtime sized and therefore must allocate the necessary memory dynamically. Consequently, it must do a full copy of the matrix $2N$ times. If this is supposed to perform efficiently, the `Matrix` class storage must be converted to `valarray<T>` to avoid the copies (the benchmark result of this approach is shown in the last row of Table 1). This shows that `valarray<T>` (or any other runtime sized data-parallel type) cannot be used as easily as `Vector<T>` in only an algorithm implementation. To avoid unnecessary memory allocations and copies, a runtime sized type always must be used pervasively in data structures and algorithms. However, as Table 1 shows, even then the compiler has a hard time generating optimal code. And the opportunities for the developer to optimize via his understanding of register usage is significantly restrained.

4.3 Data type for matrix storage

The matrix multiplication example presented above uses scalar types for data storage and SIMD types for data processing. It would be possible to build the matrix storage with SIMD types and thus eliminate the explicit load and store expressions. However, this makes the broadcasts (e.g. `a[i + n][k]`) harder to express, since a third subscript operator is required and the last index must be split into an iteration over vectors and an iteration over vector entries. Furthermore, the SIMD type storage approach makes any other operations which require access to scalars on the

matrix more complicated to express. The only conceivable abstraction for simplifying this issue is a container class that replaces the inner `std::array` which provides accessors for both scalar and vector iteration, correctly solving the inherent aliasing issue.

A ACKNOWLEDGEMENTS

This work was supported by GSI Helmholtzzentrum für Schwerionenforschung and the Hessian LOEWE initiative through the Helmholtz International Center for FAIR (HIC for FAIR).

Thanks to Lawrence Crowl for his review and questions.

B BIBLIOGRAPHY

[N3831] Robert Geva and Clark Nelson. *N3831: Language Extensions for Vector level parallelism*. ISO/IEC C++ Standards Committee Paper. 2014. URL: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3831.pdf`.

[N4184] Matthias Kretz. *N4184: SIMD Types: The Vector Type & Operations*. ISO/IEC C++ Standards Committee Paper. 2014. URL: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4184.pdf`.