Document Number: P0349R0
Date: 2016-05-24
Reply-to: Matthias Kretz <m.kretz@gsi.de>
Audience: SG1

Assumptions about the size of datapar

ABSTRACT

This paper discusses consequences of using `datapar` for architectures with variable vector width. The assumptions a compiler can make about the `size()` of `datapar` has consequences for optimizations. On the other hand, this requires restrictions on `datapar::size()` that may be surprising to users.

CONTENTS

1 INTRODUCTION

For more information on `datapar` see [P0214R0] and [1].

1.1 HARDWARE APPLICABILITY

The assumption that `datapar<T>::size()` yields a constant expression on every conceivable hardware implementation for efficient data parallel execution is limiting. There exist vector machines that advertise a variable vector length. There may be research going into future hardware to loosen the vector width on SIMD instructions. Consequently, the static member functions `static constexpr size_t size()` in `datapar` and `mask` may inhibit or at least reduce the applicability of the programming model to the widest range of available and future hardware.

1.2 OPTIMIZATION OPPORTUNITIES

Consider the following code snippet:

```
void f(const float *input, float *output, size_t N) {
  using V = datapar<float>;
  for (int i = 0; i + V::size() <= N; i += V::size()) {
    const V x = load<V>(&input[i]);
    store(x + 1, &output[i]);
  }
  // process the rest
}
```

The compiler sees a loop that reads `V::size()` elements from `input` and writes the same number of elements to `output` before going into the next iteration. The compiler traditionally would do alias analysis if it wants to determine whether it can reorder the loads and stores, as they could be dependent. With `datapar` there is an opportunity to loosen the rules. Consider that `datapar<float>` returns a different number for `size()` depending on the target system and available SIMD instruction set. The type did not require any specific number of elements. If the compiler thus were allowed to assume `V` to have any `size()`, it could assume `size() == N`. In this case it would not have to do the alias analysis and could unroll and reorder more freely.

Enabling this optimization can have a noticeable impact (depending on the target architecture, of course). Consider x86 where the `float` add instruction has a latency of 3 cycles and a throughput of 1 cycle. If the sequence of instructions must follow *load x0; x0 += 1; store x0; load x1; x1 += 1; store x1; …*, the CPU must ensure that the loads and stores of subsequent iterations are independent to enable out-of-order, pipelined, superscalar execution. Depending on the (micro)architecture, this incurs a

noticeable inefficiency. If, however, the sequence of operations can be reordered by the compiler to e.g. *load x0; load x1; x0 += 1; load x2; load x3; x1 += 1; load x4; load x5; x2 += 1; load x6; store x0; x3 += 1; store x1; ...*, then the CPU can easily pipeline the add instructions.

However, consider the following calling code (in a different TU to make things worse):

```
void g() {
  float data[1000] = ...;
  if (datapar<float>::size() <= 8) {
    f(data, data + 8, 1000 - 8);
  }
}
```

The user ensured that the load store dependencies are met. The above optimization thus would break the users intention.

2 DISCUSSION

Having a `constexpr size()` in `datapar` and `mask` for ABI parameters that are not `fixed_size<N>` thus appears to inhibit the full potential of expressing data parallelism via the type system. Is it possible to weaken the `size()` function to resolve the issues? At this point I have no solution. This paper is meant to set a starting point to either figuring out a solution or a well-informed decision to not pursue this as a design goal any further.

Considering that the inhibitor is restricted load-store reordering because of aliasing, the best and most general solution may be to enable `restrict` or something similar in C++.

2.1 NO SIZEOF

Let us start with the simple loop:

```
using V = datapar<float>;
for(int i = 0; i < N; i += V::size()) {
  store(1 + load(&input[i]), &output[i]);
}
```

If the compiler shall generate code for hardware that determines the vector width only when executing the SIMD load/store/add instructions, then `V::size()` cannot be a constant expression. Rather, the machine code would have to increment `i` using a value obtained from some register or instruction. (If `V::size()` is a constant expression, then the increment can be encoded into the instruction itself.) Would dropping `constexpr` from `size()` help? At least it would decouple the number of

elements from the type information. However, as a consequence `sizeof(V)` could also not return any useful value, since one could infer the `constexpr size()` from `sizeof`. Thus, the declaration of structures with `datapar` members could not have a useful `sizeof`.

The major advantage of types with data-parallel semantics over the control-flow approach is the vectorization of data structures. Suddenly this would not work anymore. At this point this approach does not show much appeal.

2.2 A variable ABI

Consider a special ABI tag, called `datapar_abi::variable` in the following. An instance of `datapar<T, variable>` would not have a `constexpr size()` function, no `sizeof`, and using it as a data member would be ill-formed. So what is it useable for?

```
using V = datapar<float, datapar_abi::variable>;
for (int i = 0; i < N; i += V::size()) {
  store(1 + load(&input[i]), &output[i]);
}
```

In essence this would be equivalent to writing (using some hypothetical `simd_for` loop construct, which allows vector execution of the loop body):

```
simd_for (int i = 0; i < N; ++i) {
  output[i] = 1 + input[i];
}
```

I cannot image any advantage of this construction over control-flow based vectorization except maybe the ability to mark functions as vectorizable.

In any case, calling it `datapar<T, variable>` is asking for repeating the `std::vector<bool>` mistake. Therefore, it should rather use a name such as `dyndatapar<T>`[1]

2.3 Parallel algorithms

Consider writing the loop as

```
transform(execution_policy::datapar,
          begin(input), end(input), begin(output) [](auto x) {
  return x + 1;
});
```

1 Yes, it would likely repeat many of the discussions around `dynarray`.

Here the use of `datapar` is implicit. The implementation chooses the number of elements. The user has no influence on choosing the `datapar_abi` and thus has no guarantee for the `size()` outside of the loop. (Inside the loop the code can still query `x.size()` and choose different branches.) But at this point the compiler has all necessary information to use a variable vector width or to unroll the loop and reorder loads and stores.

Is this still true if there are function calls from inside the callable? The following code can still be assumed to work correctly with an arbitrary number of elements in `datapar`, because the `Abi` parameter is unconstrained:

```
template <typename T, typename Abi>
datapar<T, Abi> f(datapar<T, Abi> x) { return x + 1; }

transform(execution_policy::datapar,
          begin(input), end(input), begin(output) [](auto x) {
  return f(x);
});
```

If, however, the `Abi` parameter is constrained, the compiler would have to be conservative. It would have to assume that the maximum number of elements it may reorder is bounded by the maximum number of elements the overloads of `g` support.

```
template <typename T>
datapar<T, datapar_abi::avx> g(datapar<T, datapar_abi::avx> x) { return x + 1; }
template <typename T>
datapar<T, datapar_abi::sse> g(datapar<T, datapar_abi::sse> x) { return x + 1; }
template <typename T>
datapar<T, datapar_abi::scalar> g(datapar<T, datapar_abi::scalar> x) { return x + 1; }

transform(execution_policy::datapar,
          begin(input), end(input), begin(output) [](auto x) {
  return g(x);
});
```

However, it is unlikely that the above code could work with all implementations of `transform`. An implementation might choose to use a `datapar<T, fixed_size<N>>`, e.g. for prologue and epilogue. Thus, the user would have to provide a `datapar<T, fixed_size<N>>` overload:

```
template <typename T, int N>
datapar<T, datapar_abi::fixed_size<N>> g(datapar<T, datapar_abi::fixed_size<N>> x) {
  return x + 1;
}
template <typename T>
datapar<T, datapar_abi::avx> g(datapar<T, datapar_abi::avx> x) { return x + 1; }
```

```
template <typename T>
datapar<T, datapar_abi::sse> g(datapar<T, datapar_abi::sse> x) { return x + 1; }
template <typename T>
datapar<T, datapar_abi::scalar> g(datapar<T, datapar_abi::scalar> x) { return x + 1; }

transform(execution_policy::datapar,
          begin(input), end(input), begin(output) [](auto x) {
  return g(x);
});
```

And suddenly the vector width is unbounded again.

3 CONCLUSION

I believe it is worthwhile to investigate the *Parallel Algorithms* optimization opportunities. From what I can tell, there would not need to be any extra wording to allow variable vector width execution and load store reordering in such cases. The ability to do these transformations follows naturally from letting the implementation choose the `datapar_abi`.

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).

B BIBLIOGRAPHY

[1] Matthias Kretz. "Extending C++ for Explicit Data-Parallel Programming via SIMD Vector Types." Frankfurt (Main), Univ. PhD thesis. 2015. DOI: 10.13140/RG.2.1.2355.4323. URL: http://publikationen.ub.uni-frankfurt.de/frontdoor/index/index/docId/38415.

[P0214R0] Matthias Kretz. *P0214R0: Data-Parallel Vector Types & Operations*. ISO/IEC C++ Standards Committee Paper. 2016. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0214r0.pdf.