Is std::mdspan a Zero-overhead Abstraction?

OLEKSANDR BACHERIKOV

Cppcon
The C++ Conference

2023

October 01 - 06

Is **`std::mdspan`** a Zero-overhead Abstraction?

Oleksandr Bacherikov
Snap Inc

What is **`std::mdspan`**?

It’s a view over a multi-dimensional array.

It’s designed primarily to be used as a function parameter.

What is the simplest multi-dimensional function?

What is the simplest multi-dimensional function?

Let’s take the smallest dimension more than 1, which is 2.

The most common 2-dimensional object is a matrix.

What is the simplest multi-dimensional function?

Let’s take the smallest dimension more than 1, which is 2.

The most common 2-dimensional object is a matrix.

Let’s take a simple operation that’s not too boring, say

matrix addition

Back to high school

Back to high school

$$A = \begin{pmatrix} -5 & 5 \\ -3 & 1 \\ 4 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 3 & 2 \\ 1 & 3 \\ -5 & 4 \end{pmatrix}$$

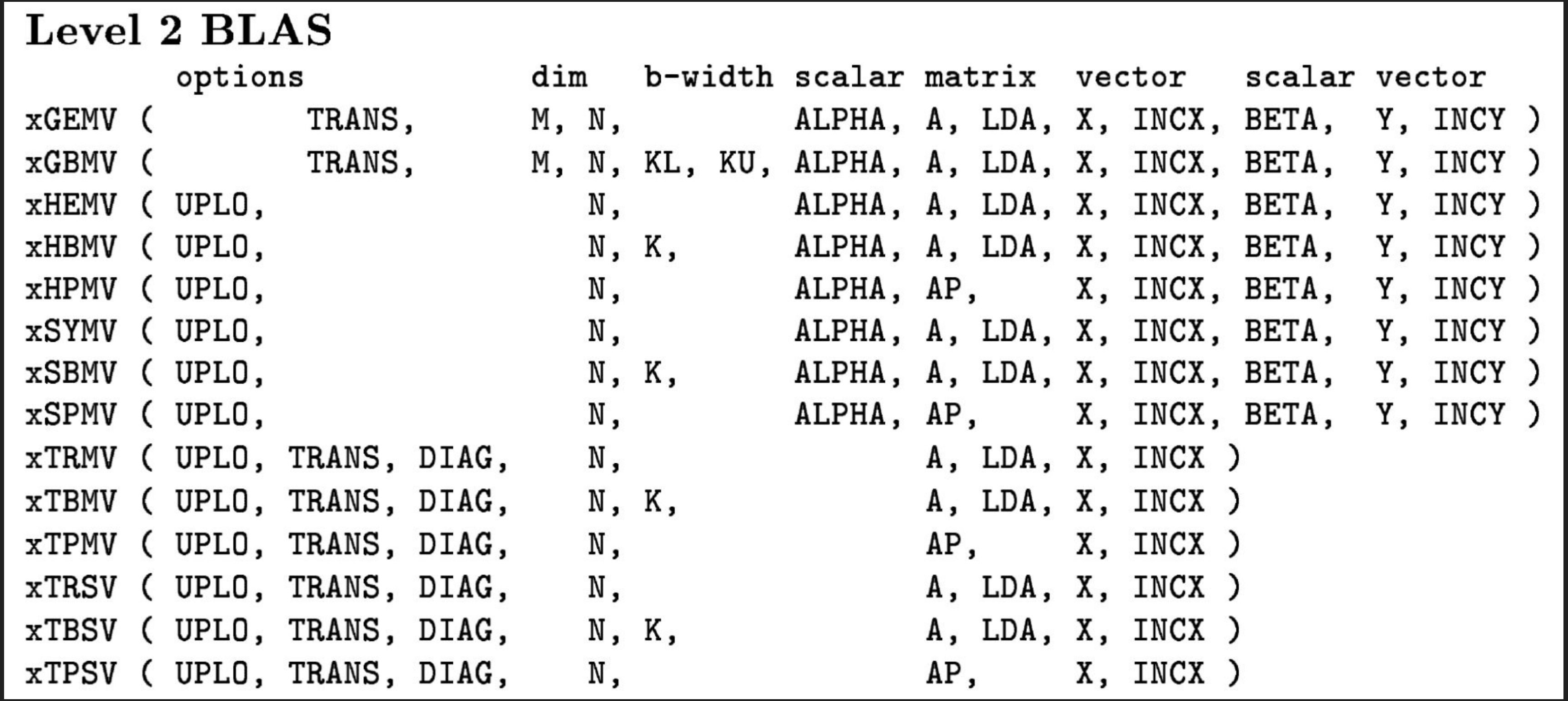
$$A + B = \begin{pmatrix} (-5 + 3) & (5 + 2) \\ (-3 + 1) & (1 + 3) \\ (4 + -5) & (0 + 4) \end{pmatrix} = \begin{pmatrix} -2 & 7 \\ -2 & 4 \\ -1 & 4 \end{pmatrix}$$

Back to C++

If the matrices are contiguous in memory, then we can treat them as ranges and simply use **`std::transform`**

```
  {-5,  5, -3,  1,  4,  0}
+
  { 3,  2,  1,  3, -5,  4}
=
  {-2,  7, -2,  4, -1,  4}
```

BLAS (Basic Linear Algebra Subprograms)



Level 2 BLAS

```
              options                dim   b-width  scalar matrix   vector    scalar vector
xGEMV (            TRANS,            M, N,          ALPHA, A, LDA, X, INCX, BETA,  Y, INCY )
xGBMV (            TRANS,            M, N, KL, KU,  ALPHA, A, LDA, X, INCX, BETA,  Y, INCY )
xHEMV ( UPLO,                           N,          ALPHA, A, LDA, X, INCX, BETA,  Y, INCY )
xHBMV ( UPLO,                           N, K,       ALPHA, A, LDA, X, INCX, BETA,  Y, INCY )
xHPMV ( UPLO,                           N,          ALPHA, AP,     X, INCX, BETA,  Y, INCY )
xSYMV ( UPLO,                           N,          ALPHA, A, LDA, X, INCX, BETA,  Y, INCY )
xSBMV ( UPLO,                           N, K,       ALPHA, A, LDA, X, INCX, BETA,  Y, INCY )
xSPMV ( UPLO,                           N,          ALPHA, AP,     X, INCX, BETA,  Y, INCY )
xTRMV ( UPLO, TRANS, DIAG,              N,                 A, LDA, X, INCX )
xTBMV ( UPLO, TRANS, DIAG,              N, K,              A, LDA, X, INCX )
xTPMV ( UPLO, TRANS, DIAG,              N,                 AP,     X, INCX )
xTRSV ( UPLO, TRANS, DIAG,              N,                 A, LDA, X, INCX )
xTBSV ( UPLO, TRANS, DIAG,              N, K,              A, LDA, X, INCX )
xTPSV ( UPLO, TRANS, DIAG,              N,                 AP,     X, INCX )
```

Matrix with a row stride

```
{-5,  5,  .,  .}
{-3,  1,  .,  .}
{ 4,  0,  .,  .}
```

BLAS-like implementation

```
void add_blas(
    size_t rows, size_t cols,
    const float* lhs, size_t lhsRowStride,
    const float* rhs, size_t rhsRowStride,
    float* dst, size_t dstRowStride
) {
    for (size_t i = 0; i < rows; ++i)
        for (size_t j = 0; j < cols; ++j)
            dst[dstRowStride * i + j] =
                lhs[lhsRowStride * i + j] +
                rhs[rhsRowStride * i + j];
}
```

```
    push    rbp
    push    r15
    push    r14
    push    r13
    push    r12
    push    rbx
    mov     qword ptr [rsp - 48], r9
    mov     qword ptr [rsp - 72], r8
    mov     qword ptr [rsp - 56], rcx
    mov     qword ptr [rsp - 64], rdx
    test    rdi, rdi
    je      .LBB0_16
    mov     rax, qword ptr [rsp + 64]
    mov     r8, qword ptr [rsp + 56]
    lea     rbx, [4*rax]
    mov     rax, qword ptr [rsp - 56]
    lea     rax, [4*rax]
    mov     qword ptr [rsp - 32], rax
    mov     rax, qword ptr [rsp - 48]
    lea     rax, [4*rax]
    mov     qword ptr [rsp - 40], rax
    mov     r14, rsi
    and     r14, -8
    mov     rax, rsi
    neg     rax
    mov     qword ptr [rsp - 8], rax
    mov     rdx, qword ptr [rsp - 64]
    lea     r13, [rdx + 16]
    mov     rcx, qword ptr [rsp - 72]
    lea     rbp, [rcx + 16]
    lea     rax, [r8 + 16]
    lea     r10, [rcx + 4]
    lea     r9, [rdx + 4]
    mov     rdx, r8
    xor     ecx, ecx
    mov     r12, r8
    mov     qword ptr [rsp - 16], rdi
    mov     qword ptr [rsp - 24], rbx
    jmp     .LBB0_3
.LBB0_2:
    inc     rcx
    mov     r8, qword ptr [rsp - 32]
    add     r13, r8
    mov     r11, qword ptr [rsp - 40]
    add     rbp, r11
    add     rax, rbx
    add     r12, rbx
```

```
    add     r10, r11
    add     r9, r8
    cmp     rcx, rdi
    je      .LBB0_16
.LBB0_3:
    test    rsi, rsi
    je      .LBB0_2
    cmp     rsi, 8
    jb      .LBB0_11
    mov     r11, rbx
    imul    r11, rcx
    add     r11, rdx
    mov     r8, qword ptr [rsp - 32]
    imul    r8, rcx
    add     r8, qword ptr [rsp - 64]
    mov     r15, r11
    sub     r15, r8
    cmp     r15, 32
    jb      .LBB0_11
    mov     r8, qword ptr [rsp - 40]
    imul    r8, rcx
    add     r8, qword ptr [rsp - 72]
    sub     r11, r8
    mov     r8d, 0
    cmp     r11, 32
    jb      .LBB0_12
    xor     r8d, r8d
.LBB0_9:
    movups  xmm0, xmmword ptr [r13 + 4*r8 - 16]
    movups  xmm1, xmmword ptr [r13 + 4*r8]
    movups  xmm2, xmmword ptr [rbp + 4*r8 - 16]
    addps   xmm2, xmm0
    movups  xmm0, xmmword ptr [rbp + 4*r8]
    addps   xmm0, xmm1
    movups  xmmword ptr [rax + 4*r8 - 16], xmm2
    movups  xmmword ptr [rax + 4*r8], xmm0
    add     r8, 8
    cmp     r14, r8
    jne     .LBB0_9
    mov     r8, r14
    cmp     r14, rsi
    je      .LBB0_2
    jmp     .LBB0_12
.LBB0_11:
    xor     r8d, r8d
.LBB0_12:
    mov     r15, r8
```

```
    test    sil, 1
    je      .LBB0_14
    mov     r11, rcx
    imul    r11, qword ptr [rsp - 56]
    mov     r15, rcx
    imul    r15, qword ptr [rsp - 48]
    mov     rdi, rcx
    imul    rdi, qword ptr [rsp + 64]
    add     r11, r8
    mov     rbx, rdx
    mov     rdx, qword ptr [rsp - 64]
    movss   xmm0, dword ptr [rdx + 4*r11]
    mov     rdx, rbx
    mov     rbx, qword ptr [rsp - 24]
    add     r15, r8
    mov     r11, qword ptr [rsp - 72]
    addss   xmm0, dword ptr [r11 + 4*r15]
    add     rdi, r8
    movss   dword ptr [rdx + 4*rdi], xmm0
    mov     rdi, qword ptr [rsp - 16]
    mov     r15, r8
    or      r15, 1
.LBB0_14:
    not     r8
    cmp     r8, qword ptr [rsp - 8]
    je      .LBB0_2
.LBB0_15:
    movss   xmm0, dword ptr [r9 + 4*r15 - 4
    addss   xmm0, dword ptr [r10 + 4*r15 - 4]
    movss   dword ptr [r12 + 4*r15], xmm0
    movss   xmm0, dword ptr [r9 + 4*r15]
    addss   xmm0, dword ptr [r10 + 4*r15]
    movss   dword ptr [r12 + 4*r15 + 4], xmm0
    add     r15, 2
    cmp     rsi, r15
    jne     .LBB0_15
    jmp     .LBB0_2
.LBB0_16:
    pop     rbx
    pop     r12
    pop     r13
    pop     r14
    pop     r15
    pop     rbp
    ret
    push    rbp
    push    r15
    push    r14
    push    r13
    push    r12
    push    rbx
    mov     qword ptr [rsp - 48], r9
    mov     qword ptr [rsp - 72], r8
    mov     qword ptr [rsp - 56], rcx
    mov     qword ptr [rsp - 64], rdx
    test    rdi, rdi
    je      .LBB0_16
    mov     rax, qword ptr [rsp + 64]
    mov     r8, qword ptr [rsp + 56]
    lea     rbx, [4*rax]
    mov     rax, qword ptr [rsp - 56]
    lea     rax, [4*rax]
    mov     qword ptr [rsp - 32], rax
    mov     rax, qword ptr [rsp - 48]
    lea     rax, [4*rax]
    mov     qword ptr [rsp - 40], rax
    mov     r14, rsi
    and     r14, -8
    mov     rax, rsi
    neg     rax
    mov     qword ptr [rsp - 8], rax
    mov     rdx, qword ptr [rsp - 64]
    lea     r13, [rdx + 16]
    mov     rcx, qword ptr [rsp - 72]
    lea     rbp, [rcx + 16]
    lea     rax, [r8 + 16]
    lea     r10, [rcx + 4]
    lea     r9, [rdx + 4]
    mov     rdx, r8
    xor     ecx, ecx
    mov     r12, r8
    mov     qword ptr [rsp - 16], rdi
    mov     qword ptr [rsp - 24], rbx
    jmp     .LBB0_3
.LBB0_2:
    inc     rcx
    mov     r8, qword ptr [rsp - 32]
    add     r13, r8
    mov     r11, qword ptr [rsp - 40]
    add     rbp, r11
    add     rax, rbx
    add     r12, rbx
```

```
    add     r10, r11
    add     r9, r8
    cmp     rcx, rdi
    je      .LBB0_16
.LBB0_3:
    test    rsi, rsi
    je      .LBB0_2
    cmp     rsi, 8
    jb      .LBB0_11
    mov     r11, rbx
    imul    r11, rcx
    add     r11, rdx
    mov     r8, qword ptr [rsp - 32]
    imul    r8, rcx
    add     r8, qword ptr [rsp - 64]
    mov     r15, r11
    sub     r15, r8
    cmp     r15, 32
    jb      .LBB0_11
    mov     r8, qword ptr [rsp - 40]
    imul    r8, rcx
    add     r8, qword ptr [rsp - 72]
    sub     r11, r8
    mov     r8d, 0
    cmp     r11, 32
    jb      .LBB0_12
    xor     r8d, r8d
.LBB0_9:
    movups  xmm0, xmmword ptr [r13 + 4*r8 - 16]
    movups  xmm1, xmmword ptr [r13 + 4*r8]
    movups  xmm2, xmmword ptr [rbp + 4*r8 - 16]
    addps   xmm2, xmm0
    movups  xmm0, xmmword ptr [rbp + 4*r8]
    addps   xmm0, xmm1
    movups  xmmword ptr [rax + 4*r8 - 16], xmm2
    movups  xmmword ptr [rax + 4*r8], xmm0
    add     r8, 8
    cmp     r14, r8
    jne     .LBB0_9
    mov     r8, r14
    cmp     r14, rsi
    je      .LBB0_2
    jmp     .LBB0_12
.LBB0_11:
    xor     r8d, r8d
.LBB0_12:
    mov     r15, r8
```

```
    test    sil, 1
    je      .LBB0_14
    mov     r11, rcx
    imul    r11, qword ptr [rsp - 56]
    mov     r15, rcx
    imul    r15, qword ptr [rsp - 48]
    mov     rdi, rcx
    imul    rdi, qword ptr [rsp + 64]
    add     r11, r8
    mov     rbx, rdx
    mov     rdx, qword ptr [rsp - 64]
    movss   xmm0, dword ptr [rdx + 4*r11]
    mov     rdx, rbx
    mov     rbx, qword ptr [rsp - 24]
    add     r15, r8
    mov     r11, qword ptr [rsp - 72]
    addss   xmm0, dword ptr [r11 + 4*r15]
    add     rdi, r8
    movss   dword ptr [rdx + 4*rdi], xmm0
    mov     rdi, qword ptr [rsp - 16]
    mov     r15, r8
    or      r15, 1
.LBB0_14:
    not     r8
    cmp     r8, qword ptr [rsp - 8]
    je      .LBB0_2
.LBB0_15:
    movss   xmm0, dword ptr [r9 + 4*r15 - 4
    addss   xmm0, dword ptr [r10 + 4*r15 - 4]
    movss   dword ptr [r12 + 4*r15], xmm0
    movss   xmm0, dword ptr [r9 + 4*r15]
    addss   xmm0, dword ptr [r10 + 4*r15]
    movss   dword ptr [r12 + 4*r15 + 4], xmm0
    add     r15, 2
    cmp     rsi, r15
    jne     .LBB0_15
    jmp     .LBB0_2
.LBB0_16:
    pop     rbx
    pop     r12
    pop     r13
    pop     r14
    pop     r15
    pop     rbp
    ret
```

Auto-vectorization in action

```
movups  xmm0, xmmword ptr [r13 + 4*r8 - 16]
movups  xmm1, xmmword ptr [r13 + 4*r8]
movups  xmm2, xmmword ptr [rbp + 4*r8 - 16]
addps   xmm2, xmm0
movups  xmm0, xmmword ptr [rbp + 4*r8]
addps   xmm0, xmm1
movups  xmmword ptr [rax + 4*r8 - 16], xmm2
movups  xmmword ptr [rax + 4*r8], xmm0
```

Vectorization is great for performance, not great for slideware.

clang16.0.4 -O1 -std=c++20

```
    push    rbx
    test    rdi, rdi
    je      .LBB0_6
    mov     rax, qword ptr [rsp + 24]
    mov     r10, qword ptr [rsp + 16]
    shl     rax, 2
    shl     r9, 2
    shl     rcx, 2
    xor     r11d, r11d
    jmp     .LBB0_2
.LBB0_5: # in Loop: Header=BB0_2 Depth=1
    inc     r11
    add     r10, rax
    add     r8, r9
    add     rdx, rcx
    cmp     r11, rdi
    je      .LBB0_6
.LBB0_2: # =>This Loop Header: Depth=1
    test    rsi, rsi
    je      .LBB0_5
    xor     ebx, ebx
.LBB0_4: # Parent Loop BB0_2 Depth=1
    movss   xmm0, dword ptr [rdx + 4*rbx]
    addss   xmm0, dword ptr [r8 + 4*rbx]
    movss   dword ptr [r10 + 4*rbx], xmm0
    inc     rbx
    cmp     rsi, rbx
    jne     .LBB0_4
    jmp     .LBB0_5
.LBB0_6:
    pop     rbx
    ret
```

Never make performance claims without measurements

rows	-O1 (ns)	-O2 (ns)	speedup
1	5.0	6.7	0.75x
2	6.8	11.4	0.60x
4	15.2	20.6	0.74x
8	28.5	47.0	0.61x
16	124.0	99.0	1.25x
32	379.0	231.0	1.64x
64	1876.0	766.0	2.45x
128	6318.0	2888.0	2.19x
256	26103.0	14319.0	1.82x

Important lessons we learned

Less assembly isn’t always better!

Vectorization isn’t always better!

Designing the `std::mdspan` version

Looks like we’re in luck, because there’s a pending proposal

P1673 “A free function linear algebra interface based on the BLAS”

which is based on `std::mdspan`.

What do we have in the proposal?

```
// [linalg.algs.blas1.add], add elementwise
template<in-object InObj1,
         in-object InObj2,
         out-object OutObj>
void add(InObj1 x,
         InObj2 y,
         OutObj z);
```

What do we have in the reference implementation?

```
template<
  class ElementType_x, class SizeType_x, size_t numRows_x, size_t numCols_x, class Layout_x, class Accessor_x,
  class ElementType_y, class SizeType_y, size_t numRows_y, size_t numCols_y, class Layout_y, class Accessor_y,
  class ElementType_z, class SizeType_z, size_t numRows_z, size_t numCols_z, class Layout_z, class Accessor_z>
void add_rank_2(
  std::experimental::mdspan<ElementType_x, std::experimental::extents<SizeType_x, numRows_x, numCols_x>, Layout_x, Accessor_x> x,
  std::experimental::mdspan<ElementType_y, std::experimental::extents<SizeType_y, numRows_y, numCols_y>, Layout_y, Accessor_y> y,
  std::experimental::mdspan<ElementType_z, std::experimental::extents<SizeType_z, numRows_z, numCols_z>, Layout_z, Accessor_z> z)
{
  static_assert(x.static_extent(0) == dynamic_extent || z.static_extent(0) == dynamic_extent || x.static_extent(0) == z.static_ext
  ...
  using size_type = std::common_type_t<SizeType_x, SizeType_y, SizeType_z>;
  for (size_type j = 0; j < x.extent(1); ++j) {
    for (size_type i = 0; i < x.extent(0); ++i) {
      z(i,j) = x(i,j) + y(i,j);
    }
  }
}
```

optimized for column-major matrices

Which layout should we use?

Our BLAS-like interface specifies one row stride, and elements are assumed to be contiguous in each each row, so the column stride is always 1.

Sound like we should use **`std::layout_stride`**, right?

Which layout should we use?

BLAS specifies one row stride, and elements are contiguous in each each row, so the column stride is always 1.

Sound like we should use **`std::layout_stride`**, right?

Wrong!

`std::layout_stride` supports only all strides specified at runtime.

If we target zero overhead, we have to specify one of the strides as 1 at compile time.

What does the Standard offer us instead?

There's a pending proposal P2642 "Padded mdspan layouts",

which proposes to write the following:

```
    mdspan<float, dextents<size_t, 2>, layout_right_padded<>>
```

"We considered a variant of layout_stride that could encode any combination of compile-time or run-time strides in the layout type. … However, we rejected this approach as overly complex for our design goals."

What does the Standard offer us instead?

There's a pending proposal P2642 "Padded mdspan layouts",

which proposes to write the following:

```
    mdspan<float, dextents<size_t, 2>, layout_right_padded<>>
```

"We considered a variant of layout_stride that could encode any combination of compile-time or run-time strides in the layout type. … However, we rejected this approach as overly complex for our design goals."

I'd still prefer to write the following

```
using layout_blas = layout_stride<extentsXX<size_t>, stridesX1<size_t>>;
    mdspan<float, layout_blas>
```

Disclaimer

In this talk I’ll be using

a slightly altered mdspan design.

Finally, matrix addition using **`mdspan`**

```cpp
void add_mdspan(
    mdspan<const float, layout_blas> lhs,
    mdspan<const float, layout_blas> rhs,
    mdspan<float, layout_blas> dst
) {
    for (size_t i = 0; i < dst.extent(0); ++i)
        for (size_t j = 0; j < dst.extent(1); ++j)
            dst(i, j) = lhs(i, j) + rhs(i, j);
}
```

Comparing the code

```
void add_blas(
    size_t rows, size_t cols,
    const float* lhs, size_t lhsRowStride,
    const float* rhs, size_t rhsRowStride,
    float* dst, size_t dstRowStride
) {
    for (size_t i = 0; i < rows; ++i)
        for (size_t j = 0; j < cols; ++j)
            dst[dstRowStride * i + j] =
                lhs[lhsRowStride * i + j] +
                rhs[rhsRowStride * i + j];
}
```

```
void add_mdspan(
    mdspan<const float, layout_blas> lhs,
    mdspan<const float, layout_blas> rhs,
    mdspan<float, layout_blas> dst
) {
    for (size_t i = 0; i < dst.extent(0); ++i)
        for (size_t j = 0; j < dst.extent(1); ++j)
            dst(i, j) = lhs(i, j) + rhs(i, j);
}
```

Comparing the code

```
void add_blas(
    size_t rows, size_t cols,
    const float* lhs, size_t lhsRowStride,
    const float* rhs, size_t rhsRowStride,
    float* dst, size_t dstRowStride
) {
    for (size_t i = 0; i < rows; ++i)
        for (size_t j = 0; j < cols; ++j)
            dst[dstRowStride * i + j] =
                lhs[lhsRowStride * i + j] +
                rhs[rhsRowStride * i + j];
}
```

```
void add_mdspan(
    mdspan<const float, layout_blas> lhs,
    mdspan<const float, layout_blas> rhs,
    mdspan<float, layout_blas> dst
) {
    for (size_t i = 0; i < dst.extent(0); ++i)
        for (size_t j = 0; j < dst.extent(1); ++j)
            dst(i, j) = lhs(i, j) + rhs(i, j);
}
```

C++ Core Guidelines, I.24: Avoid adjacent parameters that can be invoked by the same arguments in either order with different meaning.

Comparing the code

```
void add_blas(
    size_t rows, size_t cols,
    const float* lhs, size_t lhsRowStride,
    const float* rhs, size_t rhsRowStride,
    float* dst, size_t dstRowStride
) {
    for (size_t i = 0; i < rows; ++i)
        for (size_t j = 0; j < cols; ++j)
            dst[dstRowStride * i + j] =
                lhs[lhsRowStride * i + j] +
                rhs[rhsRowStride * i + j];
}
```

```
void add_mdspan(
    mdspan<const float, layout_blas> lhs,
    mdspan<const float, layout_blas> rhs,
    mdspan<float, layout_blas> dst
) {
    for (size_t i = 0; i < dst.extent(0); ++i)
        for (size_t j = 0; j < dst.extent(1); ++j)
            dst(i, j) = lhs(i, j) + rhs(i, j);
}
```

C++ Core Guidelines, I.24: Avoid adjacent parameters that can be invoked by the same arguments in either order with different meaning.

Comparing the code

```cpp
void add_blas(
    size_t rows, size_t cols,
    const float* lhs, size_t lhsRowStride,
    const float* rhs, size_t rhsRowStride,
    float* dst, size_t dstRowStride
) {
    for (size_t i = 0; i < rows; ++i)
        for (size_t j = 0; j < cols; ++j)
            dst[dstRowStride * i + j] =
                lhs[lhsRowStride * i + j] +
                rhs[rhsRowStride * i + j];
}
```

120 times worse!

```cpp
void add_mdspan(
    mdspan<const float, layout_blas> lhs,
    mdspan<const float, layout_blas> rhs,
    mdspan<float, layout_blas> dst
) {
    for (size_t i = 0; i < dst.extent(0); ++i)
        for (size_t j = 0; j < dst.extent(1); ++j)
            dst(i, j) = lhs(i, j) + rhs(i, j);
}
```

C++ Core Guidelines, I.24: Avoid adjacent parameters that can be invoked by the same arguments in either order with different meaning.

I've got bad news

```
    push    rbx
    mov     rax, qword ptr [rsp + 80]
    test    rax, rax
    je      .LBB0_6
    lea     r10, [rsp + 80]
    lea     r9, [rsp + 48]
    lea     rdi, [rsp + 16]
    mov     rcx, qword ptr [r10 + 8]
    mov     rdx, qword ptr [r10 + 16]
    mov     rsi, qword ptr [rdi + 16]
    mov     rdi, qword ptr [rdi + 24]
    mov     r8, qword ptr [r9 + 16]
    mov     r9, qword ptr [r9 + 24]
    mov     r10, qword ptr [r10 + 24]
    shl     rsi, 2
    shl     r8, 2
    shl     rdx, 2
    xor     r11d, r11d
    jmp     .LBB0_2
.LBB0_5: # in Loop: Header=BB0_2 Depth=1
    inc     r11
    add     rdi, rsi
    add     r9, r8
    add     r10, rdx
    cmp     r11, rax
    je      .LBB0_6
.LBB0_2: # =>This Loop Header: Depth=1
    test    rcx, rcx
    je      .LBB0_5
    xor     ebx, ebx
.LBB0_4: # Parent Loop BB0_2 Depth=1
    movss   xmm0, dword ptr [rdi + 4*rbx]
    addss   xmm0, dword ptr [r9 + 4*rbx]
    movss   dword ptr [r10 + 4*rbx], xmm0
    inc     rbx
    cmp     rcx, rbx
    jne     .LBB0_4
    jmp     .LBB0_5
.LBB0_6:
    pop     rbx
    ret
```

First fix idea

```
void add_blas(
    size_t rows, size_t cols,
    const float* lhs, size_t lhsRowStride,
    const float* rhs, size_t rhsRowStride,
    float* dst, size_t dstRowStride
) {
    for (size_t i = 0; i < rows; ++i)
        for (size_t j = 0; j < cols; ++j)
            dst[dstRowStride * i + j] =
                lhs[lhsRowStride * i + j] +
                rhs[rhsRowStride * i + j];
}
```

```
void add_mdspan(
    mdspan<const float, layout_blas> lhs,
    mdspan<const float, layout_blas> rhs,
    mdspan<float, layout_blas> dst
) {
    for (size_t i = 0; i < dst.extent(0); ++i)
        for (size_t j = 0; j < dst.extent(1); ++j)
            dst(i, j) = lhs(i, j) + rhs(i, j);
}
```

First fix idea

```
void add_blas(
    size_t rows, size_t cols,
    const float* lhs, size_t lhsRowStride,
    const float* rhs, size_t rhsRowStride,
    float* dst, size_t dstRowStride
) {
    for (size_t i = 0; i < rows; ++i)
        for (size_t j = 0; j < cols; ++j)
            dst[dstRowStride * i + j] =
                lhs[lhsRowStride * i + j] +
                rhs[rhsRowStride * i + j];
}
```

```
void add_mdspan(
    mdspan<const float, layout_blas> lhs,
    mdspan<const float, layout_blas> rhs,
    mdspan<float, layout_blas> dst
) {
    for (size_t i = 0; i < dst.extent(0); ++i)
        for (size_t j = 0; j < dst.extent(1); ++j)
            dst(i, j) = lhs(i, j) + rhs(i, j);
}
```

Let’s avoid passing the same extents 3 times!

`layout_stride_only`

Primary responsibility of a layout (or derived mapping in the standard) is to map a multi-dimensional index (a sequence of indices) into a single offset for the pointer.

For **`layout_stride`**, this mapping is a dot product of the multi-index and strides:

```
return ((static_cast<size_type>(i)*stride(P)) + ... + 0);
```

That is, it depends only on strides and not extents. Extents are needed only for bounds checking.

If we relax the layout concept to require only mapping, then it’s valid to have layout with strides only.

Result: extents are passed only once

```
void add_mdspan_stride_only(
    mdspan<const float, layout_stride_only<stridesX1<size_t>>> lhs,
    mdspan<const float, layout_stride_only<stridesX1<size_t>>> rhs,
    mdspan<float, layout_blas> dst
) {
    for (size_t i = 0; i < dst.extent(0); ++i)
        for (size_t j = 0; j < dst.extent(1); ++j)
            dst(i, j) = lhs(i, j) + rhs(i, j);
}
```

This gets us from 42 to 35 lines, but the target is 32

```
    push    rbx
    mov     rax, qword ptr [rsp + 16]
    test    rax, rax
    je      .LBB0_6
    lea     r10, [rsp + 16]
    mov     r8, qword ptr [r10 + 8]
    mov     r9, qword ptr [r10 + 16]
    mov     r10, qword ptr [r10 + 24]
    shl     rdi, 2
    shl     rdx, 2
    shl     r9, 2
    xor     r11d, r11d
    jmp     .LBB0_2
.LBB0_5: # in Loop: Header=BB0_2 Depth=1
    inc     r11
    add     rsi, rdi
    add     rcx, rdx
    add     r10, r9
    cmp     r11, rax
    je      .LBB0_6
.LBB0_2: # =>This Loop Header: Depth=1
    test    r8, r8
    je      .LBB0_5
    xor     ebx, ebx
.LBB0_4: # Parent Loop BB0_2 Depth=1
    movss   xmm0, dword ptr [rsi + 4*rbx]
    addss   xmm0, dword ptr [rcx + 4*rbx]
    movss   dword ptr [r10 + 4*rbx], xmm0
    inc     rbx
    cmp     r8, rbx
    jne     .LBB0_4
    jmp     .LBB0_5
.LBB0_6:
    pop     rbx
    ret
```

Do these optimizations even matter?

rows	BLAS-like	mdspan	stride_only
1	5.0	10.9	8.9
2	6.8	12.2	11.9
4	15.2	17.5	15.0
8	28.5	37.7	32.7
16	124.0	135.0	103.0
32	379.0	481.0	372.0
64	1876.0	1872.0	1559.0
128	6318.0	7365.0	6294.0
256	26103.0	27882.0	25943.0

Filling a matrix with a value

```
void fill_blas(
    size_t rows, size_t cols,
    float value,
    float* dst, size_t dstStride
) {
    for (size_t i = 0; i < rows; ++i)
        for (size_t j = 0; j < cols; ++j)
            dst[dstStride * i + j] = value;
}
```

```
void fill_mdspan(
    mdspan<float, layout_blas> dst,
    float value
) {
    for (size_t i = 0; i < dst.extent(0); ++i)
        for (size_t j = 0; j < dst.extent(1); ++j)
            dst(i, j) = value;
}
```

```
    test    rdi, rdi
    je      .LBB0_6




    shl     rcx, 2
    xor     eax, eax
    jmp     .LBB0_2
.LBB0_5: # in Loop: Header=BB0_2 Depth=1
    inc     rax
    add     rdx, rcx
    cmp     rax, rdi
    je      .LBB0_6
.LBB0_2: # =>This Loop Header: Depth=1
    test    rsi, rsi
    je      .LBB0_5
    xor     r8d, r8d
.LBB0_4: # Parent Loop BB0_2 Depth=1
    movss   dword ptr [rdx + 4*r8], xmm0
    inc     r8
    cmp     rsi, r8
    jne     .LBB0_4
    jmp     .LBB0_5
.LBB0_6:
    ret
```

BLAS-like

```
    mov     rax, qword ptr [rsp + 8]
    test    rax, rax
    je      .LBB0_6
    lea     rsi, [rsp + 8]
    mov     rcx, qword ptr [rsi + 8]
    mov     rdx, qword ptr [rsi + 16]
    mov     rsi, qword ptr [rsi + 24]
    shl     rdx, 2
    xor     edi, edi
    jmp     .LBB0_2
.LBB0_5: # in Loop: Header=BB0_2 Depth=1
    inc     rdi
    add     rsi, rdx
    cmp     rdi, rax
    je      .LBB0_6
.LBB0_2: # =>This Loop Header: Depth=1
    test    rcx, rcx
    je      .LBB0_5
    xor     r8d, r8d
.LBB0_4: # Parent Loop BB0_2 Depth=1
    movss   dword ptr [rsi + 4*r8], xmm0
    inc     r8
    cmp     rcx, r8
    jne     .LBB0_4
    jmp     .LBB0_5
.LBB0_6:
    ret
```

`mdspan`

```
    test    rdi, rdi
    je      .LBB0_6
```

rsp is the register for the stack pointer

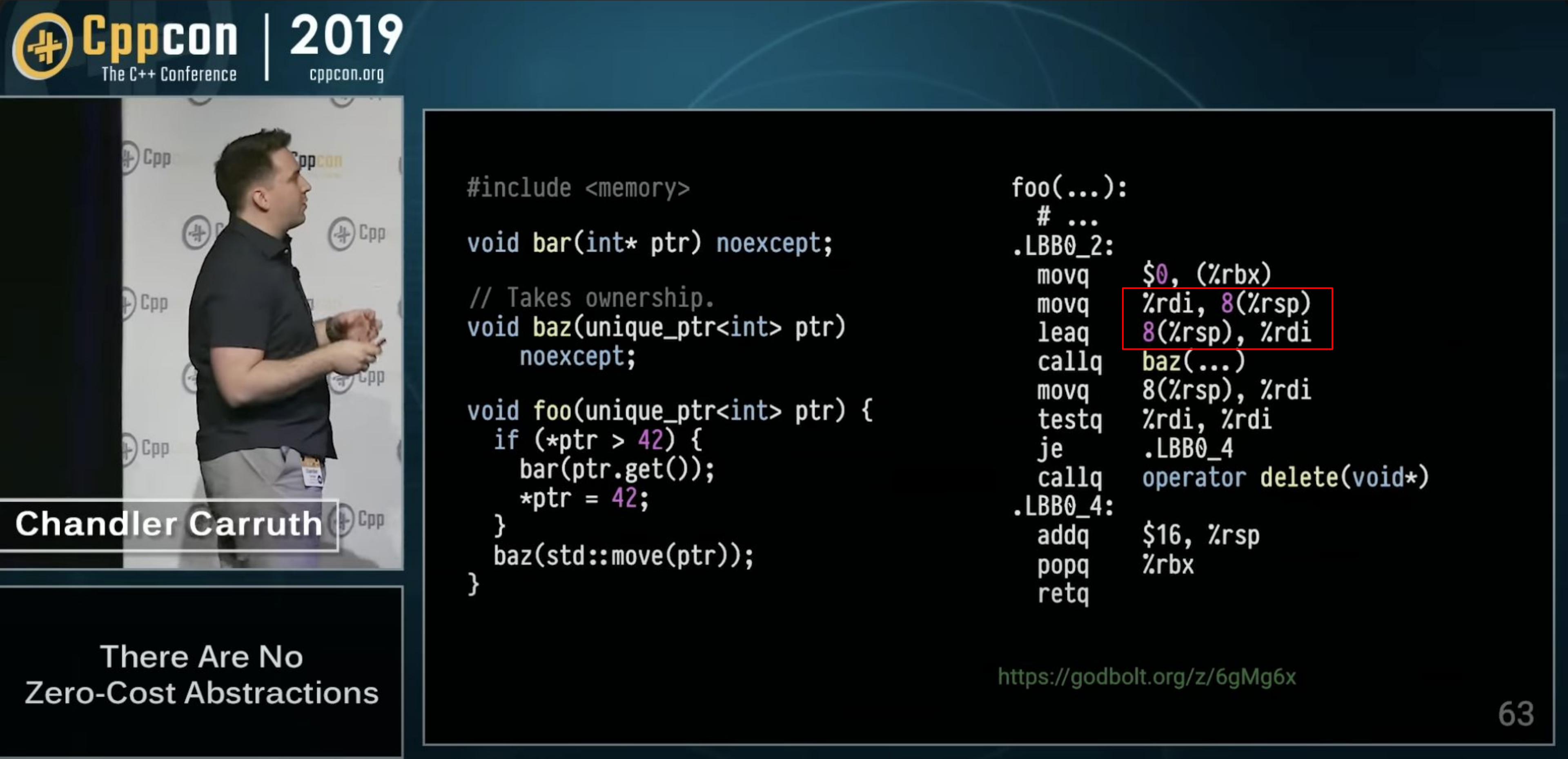
```
    shl     rcx, 2
    xor     eax, eax
    jmp     .LBB0_2
.LBB0_5: # in Loop: Header=BB0_2 Depth=1
    inc     rax
    add     rdx, rcx
    cmp     rax, rdi
    je      .LBB0_6
.LBB0_2: # =>This Loop Header: Depth=1
    test    rsi, rsi
    je      .LBB0_5
    xor     r8d, r8d
.LBB0_4: # Parent Loop BB0_2 Depth=1
    movss   dword ptr [rdx + 4*r8], xmm0
    inc     r8
    cmp     rsi, r8
    jne     .LBB0_4
    jmp     .LBB0_5
.LBB0_6:
    ret
```

BLAS-like

```
    mov     rax, qword ptr [rsp + 8]
    test    rax, rax
    je      .LBB0_6
    lea     rsi, [rsp + 8]
    mov     rcx, qword ptr [rsi + 8]
    mov     rdx, qword ptr [rsi + 16]
    mov     rsi, qword ptr [rsi + 24]
    shl     rdx, 2
    xor     edi, edi
    jmp     .LBB0_2
.LBB0_5: # in Loop: Header=BB0_2 Depth=1
    inc     rdi
    add     rsi, rdx
    cmp     rdi, rax
    je      .LBB0_6
.LBB0_2: # =>This Loop Header: Depth=1
    test    rcx, rcx
    je      .LBB0_5
    xor     r8d, r8d
.LBB0_4: # Parent Loop BB0_2 Depth=1
    movss   dword ptr [rsi + 4*r8], xmm0
    inc     r8
    cmp     rcx, r8
    jne     .LBB0_4
    jmp     .LBB0_5
.LBB0_6:
    ret
```

mdspan

Can Chandler save us?



```
#include <memory>

void bar(int* ptr) noexcept;

// Takes ownership.
void baz(unique_ptr<int> ptr)
    noexcept;

void foo(unique_ptr<int> ptr) {
  if (*ptr > 42) {
    bar(ptr.get());
    *ptr = 42;
  }
  baz(std::move(ptr));
}
```

```
foo(...):
  # ...
.LBB0_2:
  movq    $0, (%rbx)
  movq    %rdi, 8(%rsp)
  leaq    8(%rsp), %rdi
  callq   baz(...)
  movq    8(%rsp), %rdi
  testq   %rdi, %rdi
  je      .LBB0_4
  callq   operator delete(void*)
.LBB0_4:
  addq    $16, %rsp
  popq    %rbx
  retq
```

https://godbolt.org/z/6gMg6x

63

I’ve got bad and good news for you

I’ve got bad and good news for you

Bad news: that comment doesn’t help

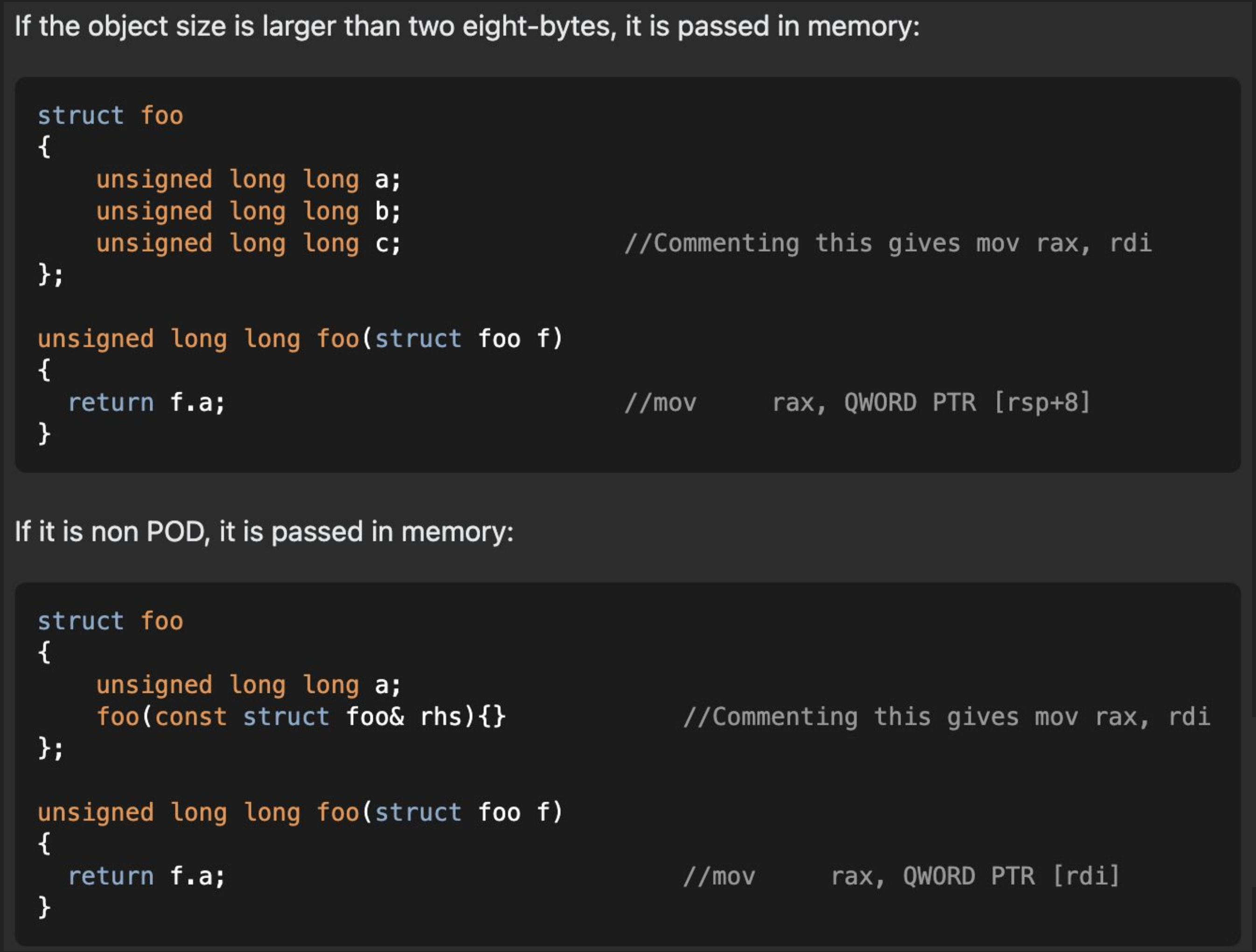
```
static_assert(
    std::is_trivially_copy_constructible_v<mdspan<float, layout_blas>>);
static_assert(
    std::is_trivially_move_constructible_v<mdspan<float, layout_blas>>);
static_assert(std::is_trivially_destructible_v<mdspan<float, layout_blas>>);
static_assert(std::is_trivial_v<layout_blas>);
static_assert(
    sizeof(mdspan<float, layout_blas>) ==
    sizeof(float*) + 3 * sizeof(size_t));
```

I’ve got bad and good news for you

Bad news: that comment doesn’t help

```
static_assert(
    std::is_trivially_copy_constructible_v<mdspan<float, layout_blas>>);
static_assert(
    std::is_trivially_move_constructible_v<mdspan<float, layout_blas>>);
static_assert(std::is_trivially_destructible_v<mdspan<float, layout_blas>>);
static_assert(std::is_trivial_v<layout_blas>);
static_assert(
    sizeof(mdspan<float, layout_blas>) ==
    sizeof(float*) + 3 * sizeof(size_t));
```

Good news: we’ll learn something new today!



If the object size is larger than two eight-bytes, it is passed in memory:

```
struct foo
{
    unsigned long long a;
    unsigned long long b;
    unsigned long long c;               //Commenting this gives mov rax, rdi
};

unsigned long long foo(struct foo f)
{
  return f.a;                           //mov     rax, QWORD PTR [rsp+8]
}
```

If it is non POD, it is passed in memory:

```
struct foo
{
    unsigned long long a;
    foo(const struct foo& rhs){}            //Commenting this gives mov rax, rdi
};

unsigned long long foo(struct foo f)
{
  return f.a;                               //mov     rax, QWORD PTR [rdi]
}
```

[C++ on x86-64: when are structs/classes passed and returned in registers?]

answered Feb 23, 2017 at 10:45

Margaret Bloom

41.9k ● 5 ● 78 ● 124

Solution

```
void fill_mdspan_split(
    extentsXX<size_t> extents,
    mdspan<float, layout_stride_only<stridesX1<size_t>>> dst,
    float value
) {
    for (size_t i = 0; i < extents[0]; ++i)
        for (size_t j = 0; j < extents[1]; ++j)
            dst(i, j) = value;
}
```

We did it!

```
    test    rdi, rdi
    je      .LBB0_6
    shl     rcx, 2
    xor     eax, eax
    jmp     .LBB0_2
.LBB0_5: # in Loop: Header=BB0_2 Depth=1
    inc     rax
    add     rdx, rcx
    cmp     rax, rdi
    je      .LBB0_6
.LBB0_2: # =>This Loop Header: Depth=1
    test    rsi, rsi
    je      .LBB0_5
    xor     r8d, r8d
.LBB0_4: # Parent Loop BB0_2 Depth=1
    movss   dword ptr [rdx + 4*r8], xmm0
    inc     r8
    cmp     rsi, r8
    jne     .LBB0_4
    jmp     .LBB0_5
.LBB0_6:
    ret
```

```
    test    rdi, rdi
    je      .LBB0_6
    shl     rdx, 2
    xor     eax, eax
    jmp     .LBB0_2
.LBB0_5: # in Loop: Header=BB0_2 Depth=1
    inc     rax
    add     rcx, rdx
    cmp     rax, rdi
    je      .LBB0_6
.LBB0_2: # =>This Loop Header: Depth=1
    test    rsi, rsi
    je      .LBB0_5
    xor     r8d, r8d
.LBB0_4: # Parent Loop BB0_2 Depth=1
    movss   dword ptr [rcx + 4*r8], xmm0
    inc     r8
    cmp     rsi, r8
    jne     .LBB0_4
    jmp     .LBB0_5
.LBB0_6:
    ret
```

Going back to our addition example

We can apply the same approach and avoid extra stack access.

Going back to our addition example

We can apply the same approach and avoid extra stack access.

But then we can notice something interesting.

Assembly for BLAS-like implementation

```
    push    rbx
    test    rdi, rdi
    je      .LBB0_6
    mov     rax, qword ptr [rsp + 24]
    mov     r10, qword ptr [rsp + 16]
    shl     rax, 2
    shl     r9, 2
    shl     rcx, 2
    xor     r11d, r11d
    jmp     .LBB0_2
.LBB0_5: # in Loop: Header=BB0_2 Depth=1
    inc     r11
    add     r10, rax
    add     r8, r9
    add     rdx, rcx
    cmp     r11, rdi
    je      .LBB0_6
.LBB0_2: # =>This Loop Header: Depth=1
    test    rsi, rsi
    je      .LBB0_5
    xor     ebx, ebx
.LBB0_4: # Parent Loop BB0_2 Depth=1
    movss   xmm0, dword ptr [rdx + 4*rbx]
    addss   xmm0, dword ptr [r8 + 4*rbx]
    movss   dword ptr [r10 + 4*rbx], xmm0
    inc     rbx
    cmp     rsi, rbx
    jne     .LBB0_4
    jmp     .LBB0_5
.LBB0_6:
    pop     rbx
    ret
```

System V AMD64 ABI

“Parameters to functions are passed in the registers **`rdi,rsi,rdx,rcx,r8,r9`**, and further values are passed on the stack in reverse order.”

https://wiki.osdev.org/System_V_ABI

That’s 6 registers for parameters.

Our function has 8 register-size parameters

```
void add_blas(
    size_t rows, size_t cols,
    const float* lhs, size_t lhsRowStride,
    const float* rhs, size_t rhsRowStride,
    float* dst, size_t dstRowStride
) {
    for (size_t i = 0; i < rows; ++i)
        for (size_t j = 0; j < cols; ++j)
            dst[dstRowStride * i + j] =
                lhs[lhsRowStride * i + j] +
                rhs[rhsRowStride * i + j];
}
```

Smaller ints benefit stack usage

```
void add_blas_extents(
    extentsXX<uint32_t> extents,
    const float* lhs, size_t lhsRowStride,
    const float* rhs, size_t rhsRowStride,
    float* dst, size_t dstRowStride
) {
    for (uint32_t i = 0; i < extents[0]; ++i)
        for (uint32_t j = 0; j < extents[1]; ++j)
            dst[dstRowStride * i + j] =
                lhs[lhsRowStride * i + j] +
                rhs[rhsRowStride * i + j];
}
```

Revisiting user API

Improving performance is good.

But making the user manually split their multi-dimensional arrays into pieces just to use our function isn’t good.

I don’t know what’s the best solution.

I have one idea, maybe someone will have a better one.

Inlining wrapper

```
template<class MDDst, class MDLhs, class MDRhs>
    requires (/* element type and compile-time extents match */)
__attribute__((__always_inline__))
void add(MDDst&& dst, MDLhs&& lhs, MDRhs&& rhs) {
    assert(lhs.extents() == rhs.extents());
    if constexpr (requires { dst.resize(lhs.extents()); })
        dst.resize(lhs.extents());
    else
        assert(dst.extents() == lhs.extents());
    /* call our optimized implementation */
}
```

Expression templates

The user should be able to simply write

```
    dst = lhs + rhs;
```

Which internally first would create an expression template for **`lhs + rhs`**, then call the **`add`** function on assignment.

Microsoft x64 calling convention

“The x64 ABI uses a four-register fast-call calling convention by default. … Integer arguments are passed in registers **`RCX, RDX, R8, and R9`**.”

“Any argument that doesn't fit in 8 bytes, or isn't 1, 2, 4, or 8 bytes, must be passed by reference.”

https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention

Microsoft x64 calling convention

“The x64 ABI uses a four-register fast-call calling convention by default. … Integer arguments are passed in registers **`RCX, RDX, R8, and R9`**.”

“Any argument that doesn't fit in 8 bytes, or isn't 1, 2, 4, or 8 bytes, must be passed by reference.”

https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention

It looks like Microsoft just doesn’t like high-level abstractions.

BLAS-like

```
rows$ = 8
cols$ = 16
value$ = 24
dst$ = 32
dstRowStride$ = 40
void fill_blas(unsigned __int64,unsigned
__int64,float,float *,unsigned __int64)
$LN19:
    test    rcx, rcx
    je      SHORT $LN17@fill_blas
    mov     QWORD PTR [rsp+8], rdi
    mov     rax, QWORD PTR dstRowStride$[rsp]
    mov     r8, rcx
    movss   DWORD PTR [rsp+24], xmm2
    lea     r10, QWORD PTR [rax*4]
$LL4@fill_blas:
    test    rdx, rdx
    je      SHORT $LN2@fill_blas
    movsxd  rax, DWORD PTR value$[rsp]
    mov     rdi, r9
    mov     rcx, rdx
    rep stosd
$LN2@fill_blas:
    add     r9, r10
    sub     r8, 1
    jne     SHORT $LL4@fill_blas
    mov     rdi, QWORD PTR [rsp+8]
$LN17@fill_blas:
    ret     0
```

mdspan

```
extents$ = 8
dst$ = 16
value$ = 24
void fill_mdspan_split(extents,mdspan,float)
$LN20:
    mov     QWORD PTR [rsp+8], rdi
    mov     r9, QWORD PTR [rcx]
    xor     r8d, r8d
    movss   DWORD PTR [rsp+24], xmm2
    test    r9, r9
    je      SHORT $LN3@fill_mdspa
    mov     r10, QWORD PTR [rcx+8]
$LL4@fill_mdspa:
    test    r10, r10
    je      SHORT $LN2@fill_mdspa
    mov     rax, QWORD PTR [rdx]
    mov     rcx, r8
    imul    rcx, QWORD PTR [rdx+8]
    lea     rdi, QWORD PTR [rax+rcx*4]
    movsxd  rax, DWORD PTR value$[rsp]
    mov     rcx, r10
    rep stosd
$LN2@fill_mdspa:
    inc     r8
    cmp     r8, r9
    jb      SHORT $LL4@fill_mdspa
$LN3@fill_mdspa:
    mov     rdi, QWORD PTR [rsp+8]
    ret     0
```

Conclusions

- Doing the best thing is much more difficult than it should be.

Conclusions

- Doing the best thing is much more difficult than it should be.
- If you want parameters to be passed in registers, make sure they are trivial to copy and small enough (8 bytes on Windows, 16 on other major platforms).
- This might mean sticking to raw pointers and ints on the low level.
Maybe, we can create `mdspan`’s from them inside the implementation.
- Don’t pass unnecessary large parameters, because registers for them are limited (4 on Windows, 6 otherwise).

Conclusions

- Doing the best thing is much more difficult than it should be.
- If you want parameters to be passed in registers, make sure they have trivial copy and small enough (8 bytes on Windows, 16 on other major platforms).
- This might mean sticking to raw pointers and ints on the low level.
  Maybe, we can create **`mdspan`**’s from them inside the implementation.
- Don’t pass unnecessary large parameters, because registers for them are limited (4 on Windows, 6 otherwise).
- Convenient wrappers can hide this complexity from the end user.

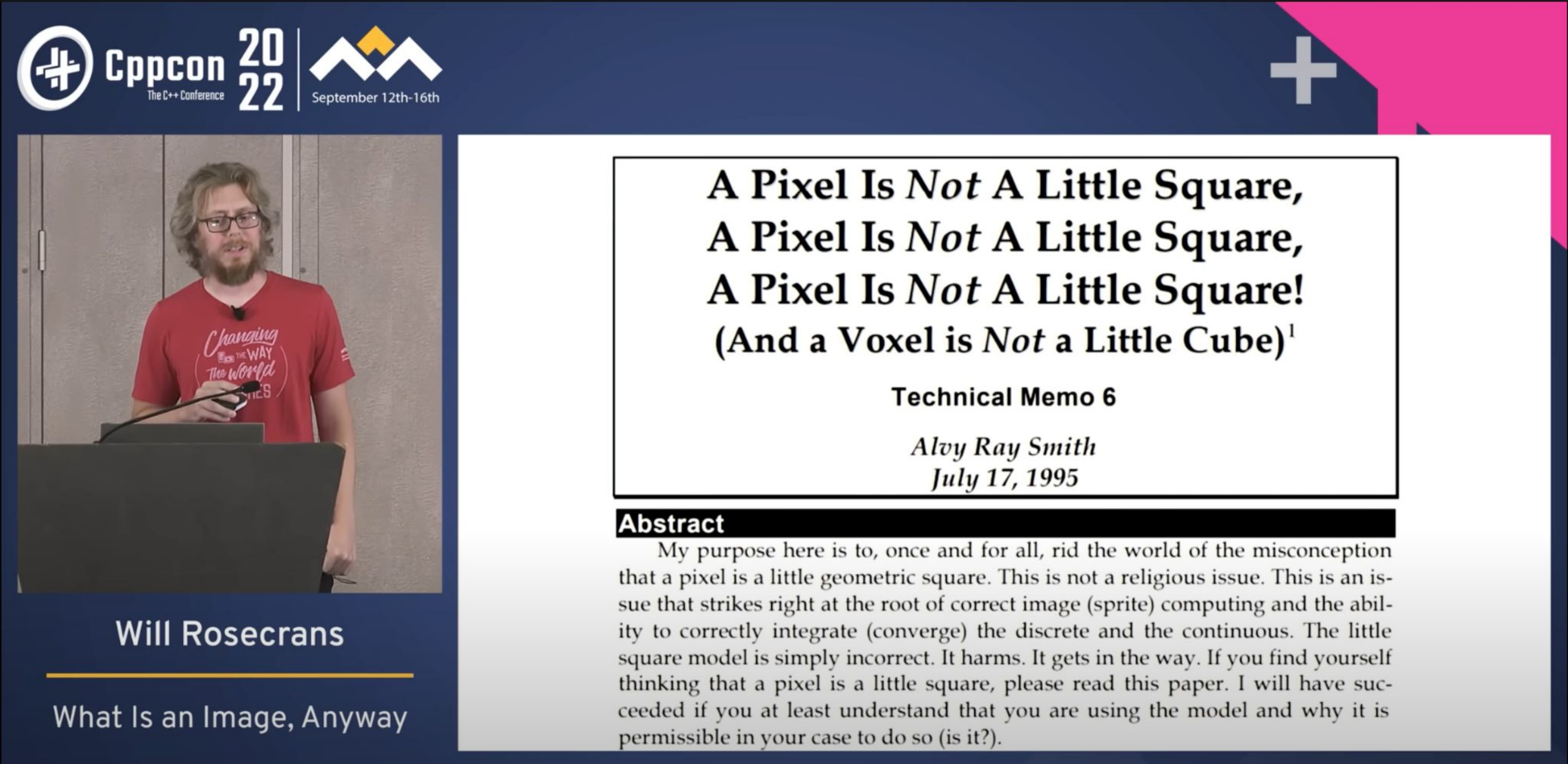
Conclusions

- Doing the best thing is much more difficult than it should be.
- If you want parameters to be passed in registers, make sure they have trivial copy and small enough (8 bytes on Windows, 16 on other major platforms).
- This might mean sticking to raw pointers and ints on the low level.
  Maybe, we can create **`mdspan`**’s from them inside the implementation.
- Don’t pass unnecessary large parameters, because registers for them are limited (4 on Windows, 6 otherwise).
- Convenient wrappers can hide this complexity from the user.
- **`std::mdspan`** isn’t flexible enough, and it may be too late to fix it.

Thank you for attention!

To higher dimensions

What is an image?



A Pixel Is *Not* A Little Square,
A Pixel Is *Not* A Little Square,
A Pixel Is *Not* A Little Square!
(And a Voxel is *Not* a Little Cube)[1]

Technical Memo 6

Alvy Ray Smith
July 17, 1995

Abstract

My purpose here is to, once and for all, rid the world of the misconception that a pixel is a little geometric square. This is not a religious issue. This is an issue that strikes right at the root of correct image (sprite) computing and the ability to correctly integrate (converge) the discrete and the continuous. The little square model is simply incorrect. It harms. It gets in the way. If you find yourself thinking that a pixel is a little square, please read this paper. I will have succeeded if you at least understand that you are using the model and why it is permissible in your case to do so (is it?).

Image representation

Two different approaches:

1. Two-dimensional array of pixels.
2. Three-dimensional array of scalars,

   where the last dimension is interpreted as a pixel,

   which has a specified color space.

Benefits of a three-dimensional representation

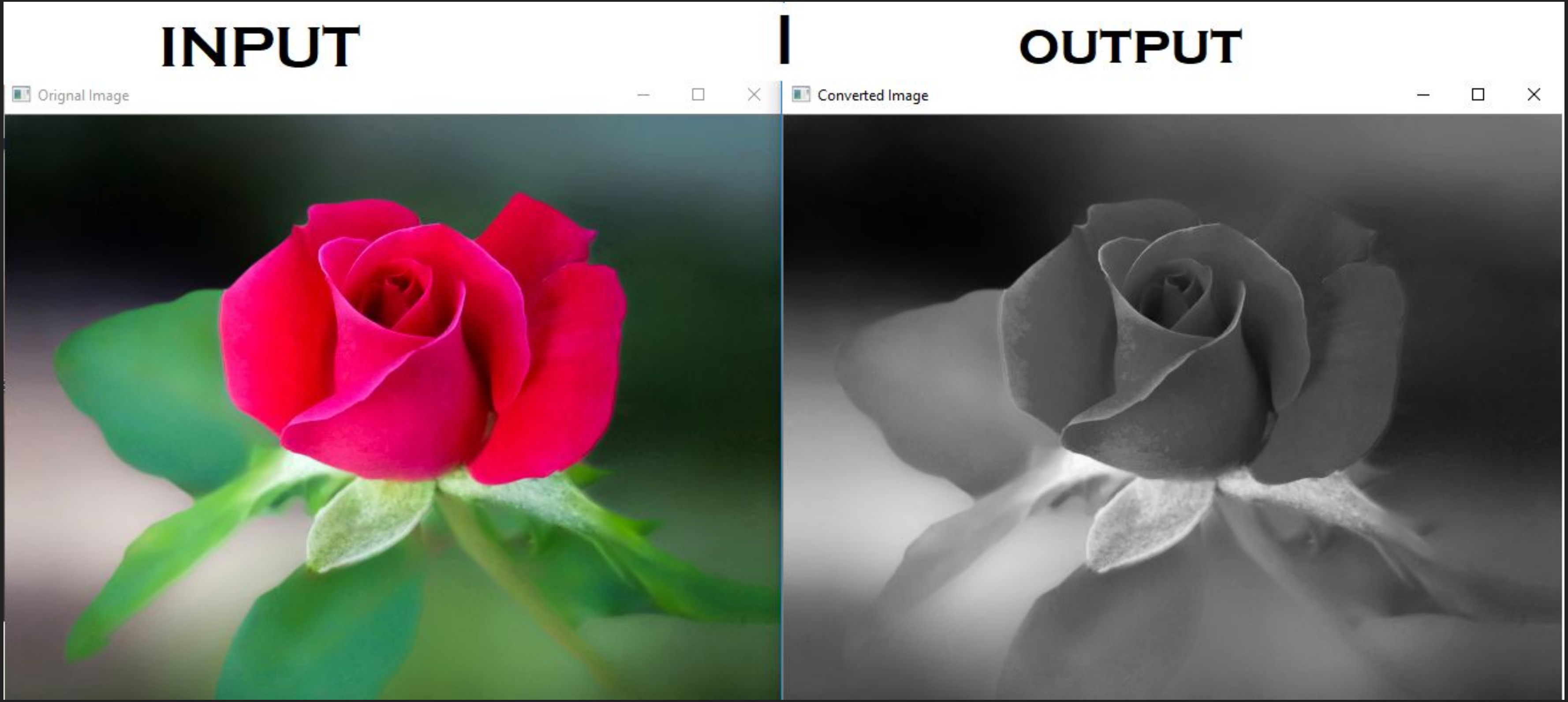
- A lot image operations don’t care about the pixel color space, only its size and scalar type:
  - slicing, copy
  - alpha blending
- For such operations, we can avoid generating identical assembly for different color spaces, without relying on non-standard Identical Code Folding.
- It allows to formulate in-place color conversions, for example, from RGB to BGR.
- Good as an example for this presentation.

Image layout draft

```
template<class ColorSpace>
struct ImageLayout {
    size_t rows;
    size_t cols;
    size_t rowStride;
    [[no_unique_address]] ColorSpace colorSpace;

    auto extents() const {
        return {rows, cols, channelCount(colorSpace)};
    }
    auto strides() const {
        return {rowStride, channelCount(colorSpace),
                std::integral_constant<size_t, 1>{}};
    }
};
```

RGB to Grayscale conversion



Color spaces

```cpp
struct RGB {};

auto channelCount(RGB) {
    return std::integral_constant<size_t, 3>{};
}

struct Grayscale {};

auto channelCount(Grayscale) {
    return std::integral_constant<size_t, 1>{};
}
```

Conversion using **`mdspan`** (draft)

```cpp
void convertColorSpace(
    mdspan<uint8_t, ImageLayout<Grayscale>> dst,
    mdspan<const uint8_t, ImageLayout<RGB>> src
) {
    for (size_t i = 0; i < dst.extent(0); ++i) {
        for (size_t j = 0; j < dst.extent(1); ++j) {
            auto srcPixel = src(i, j);
            dst(i, j, 0) =
                srcPixel[0] * 0.299f +
                srcPixel[1] * 0.587f +
                srcPixel[2] * 0.114f;
        }
    }
}
```

Thank you again!