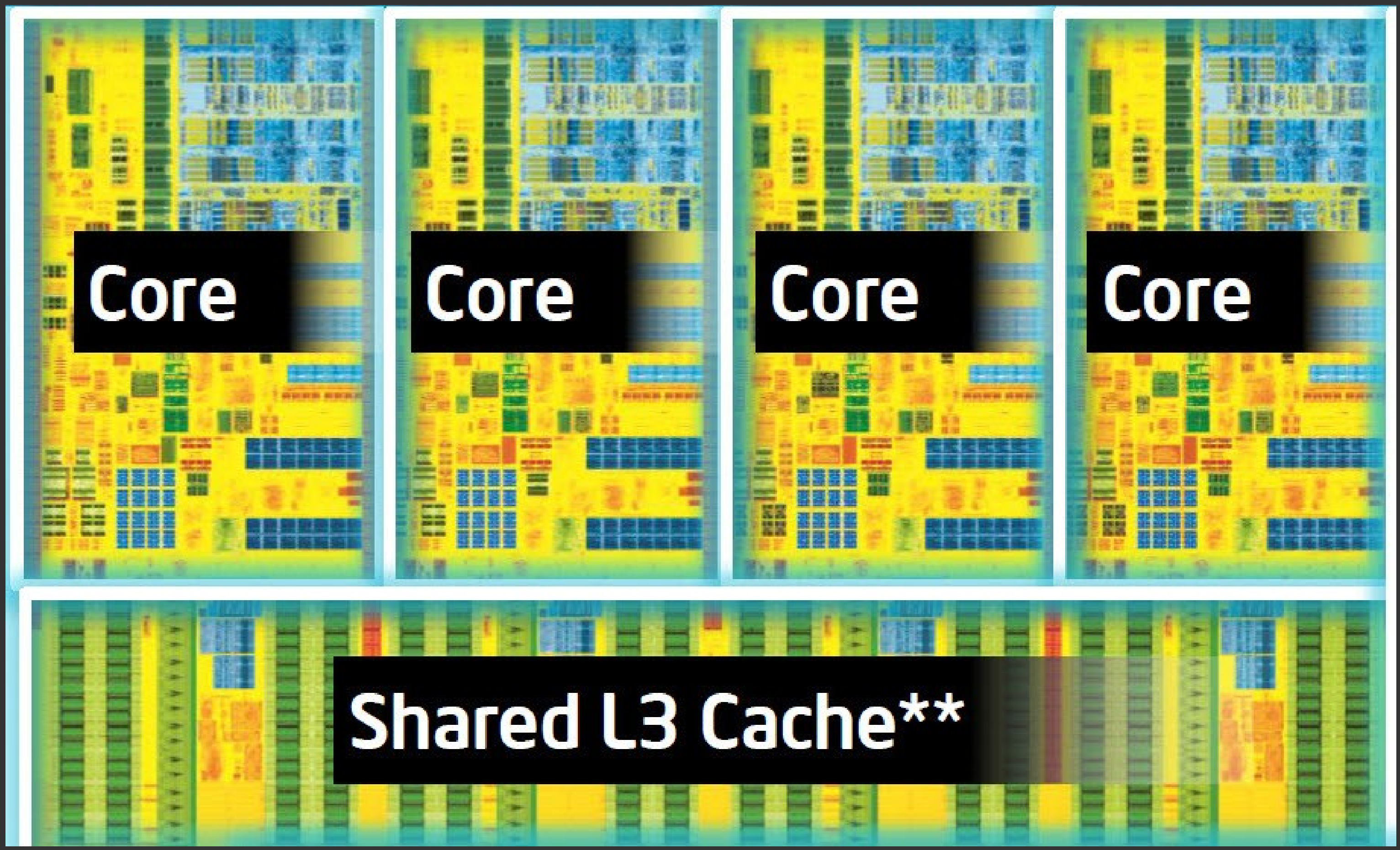
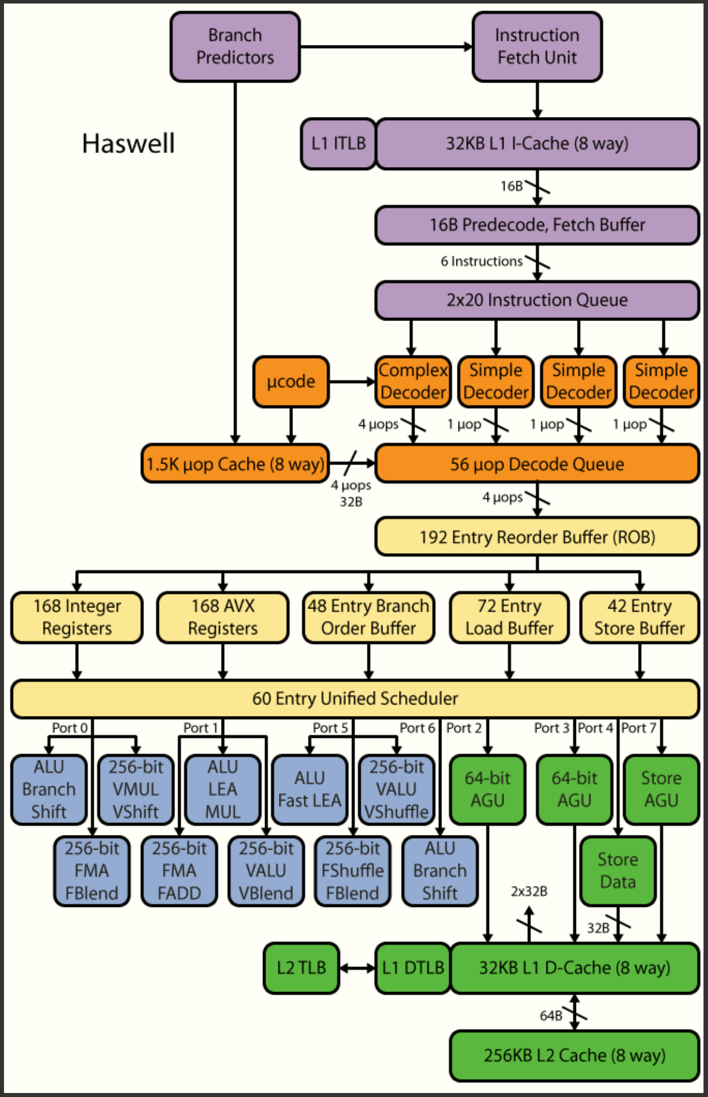
Utilizing the other 80% of your system's performance: Starting with Vectorization

Ulrich Drepper
drepper@gmail.com

Parallelism: Sure, it's covered!

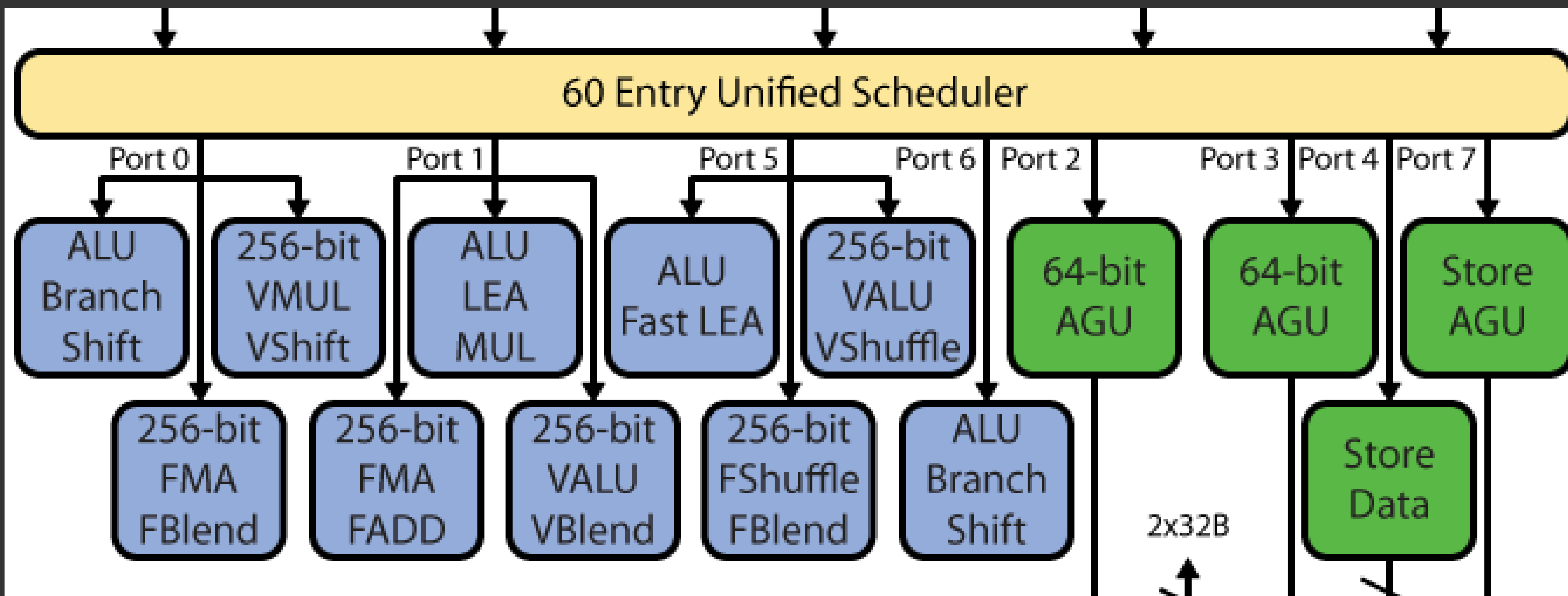


Also growing: vectors



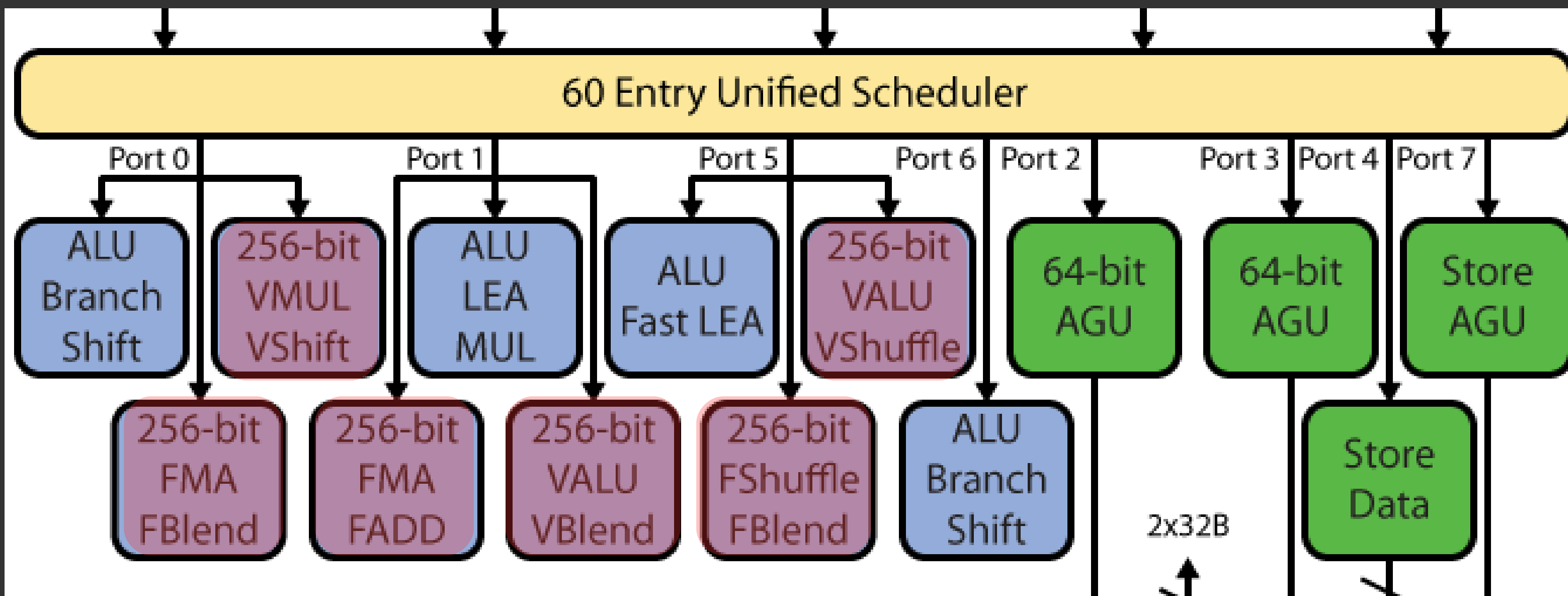
Similarly for AMD, Arm, …

EX in details



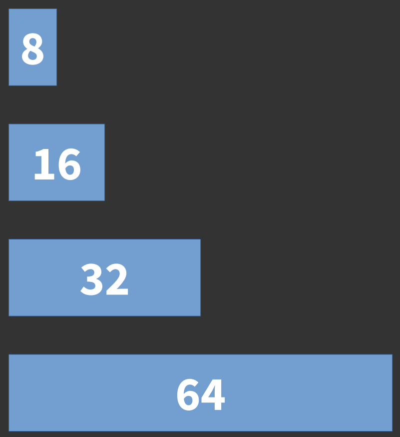
Similarly for AMD, Arm, …

EX in details



Similarly for AMD, Arm, ...

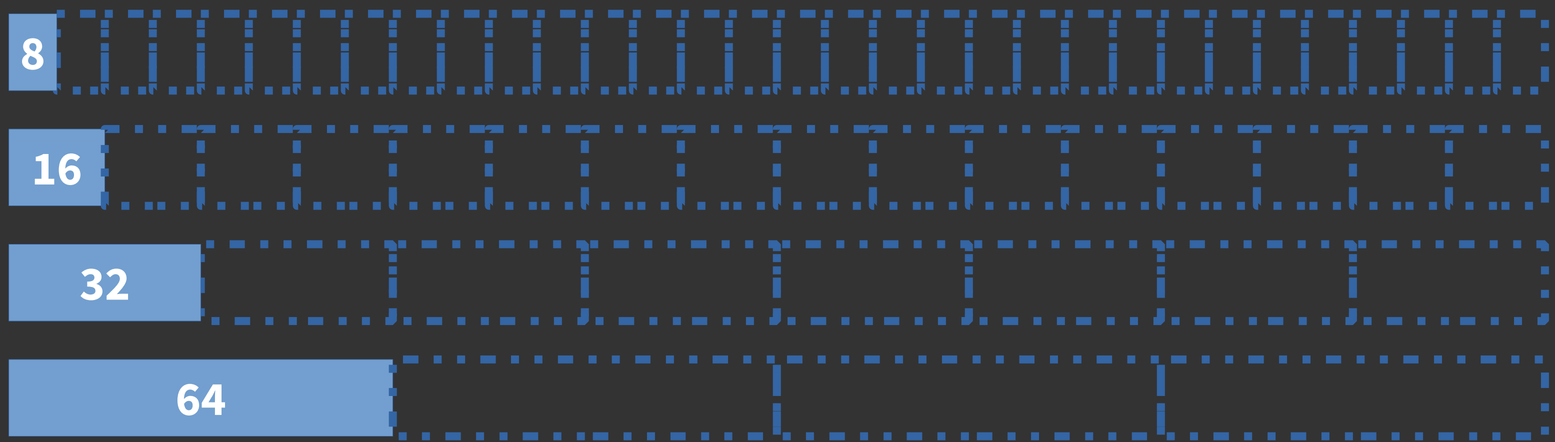
Vector Register Use



Integer: 8-, 16-, 32-, and 64-bit

FP: 16-, 32-, and 64-bit

Vector Register Use



Integer: 8-, 16-, 32-, and 64-bit: **97%, 94%, 88%, and 75% unused**

FP: 16-, 32-, and 64-bit: **94%, 88%, and 75% unused**

Marketing

- FLOPS computation

#Cores x Clock x #Ops/Cycle x VecSize/32bit x 2

256b now 512b soon

or 16bit if supported

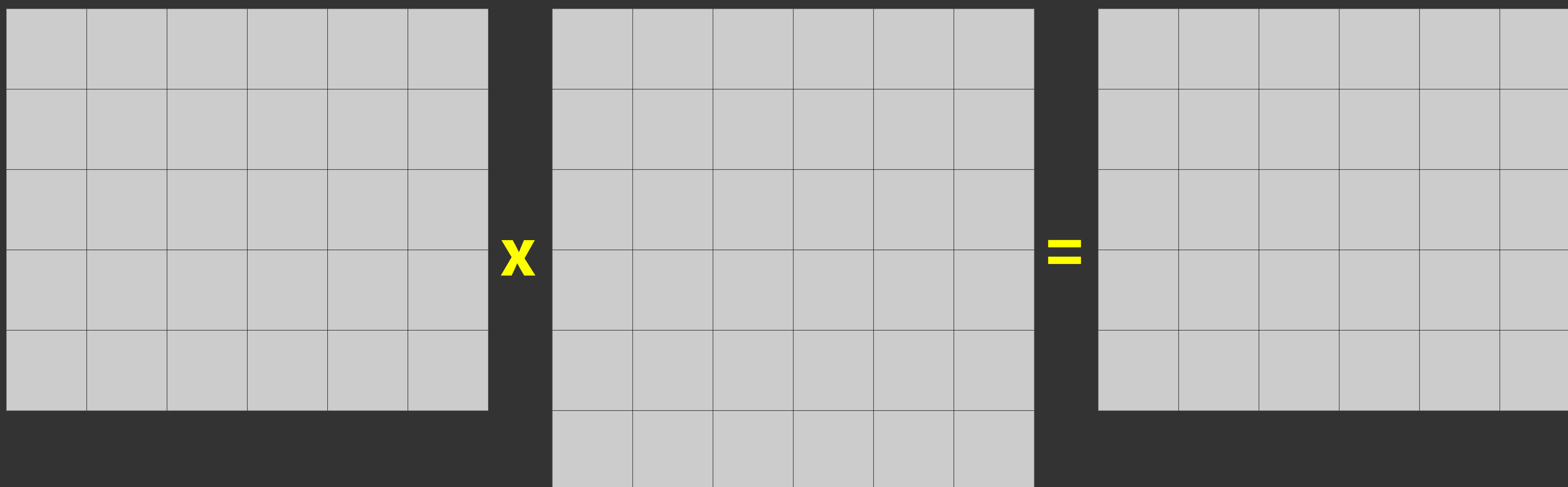
For FMA

What can be vectorized?

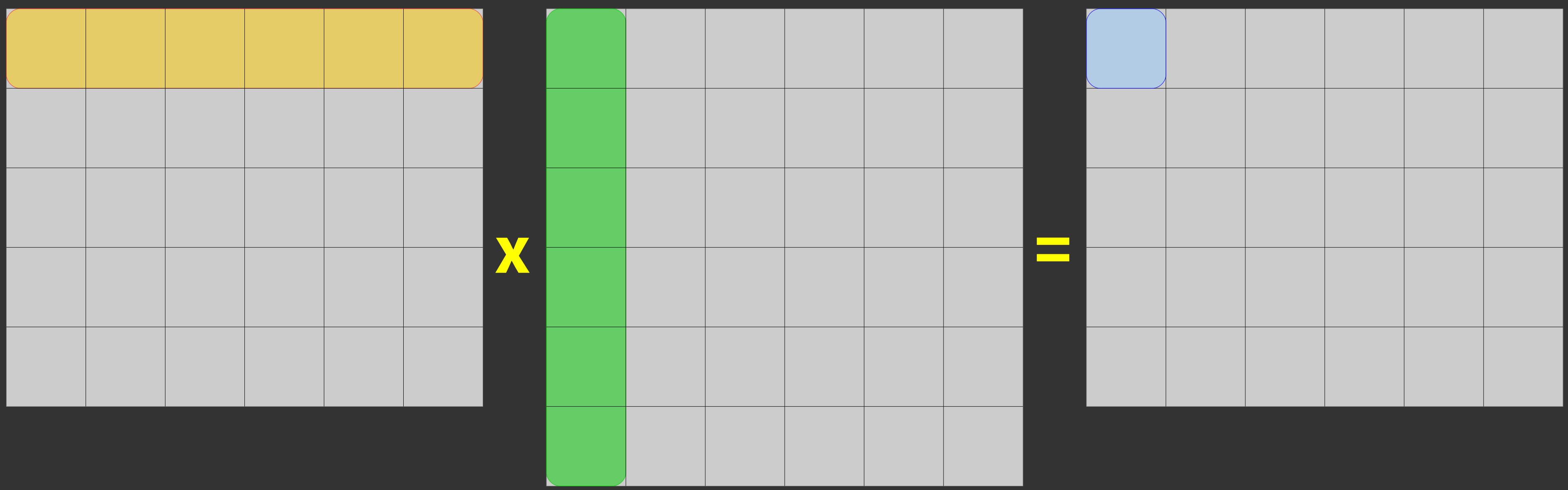
- Relatively simple:

  math

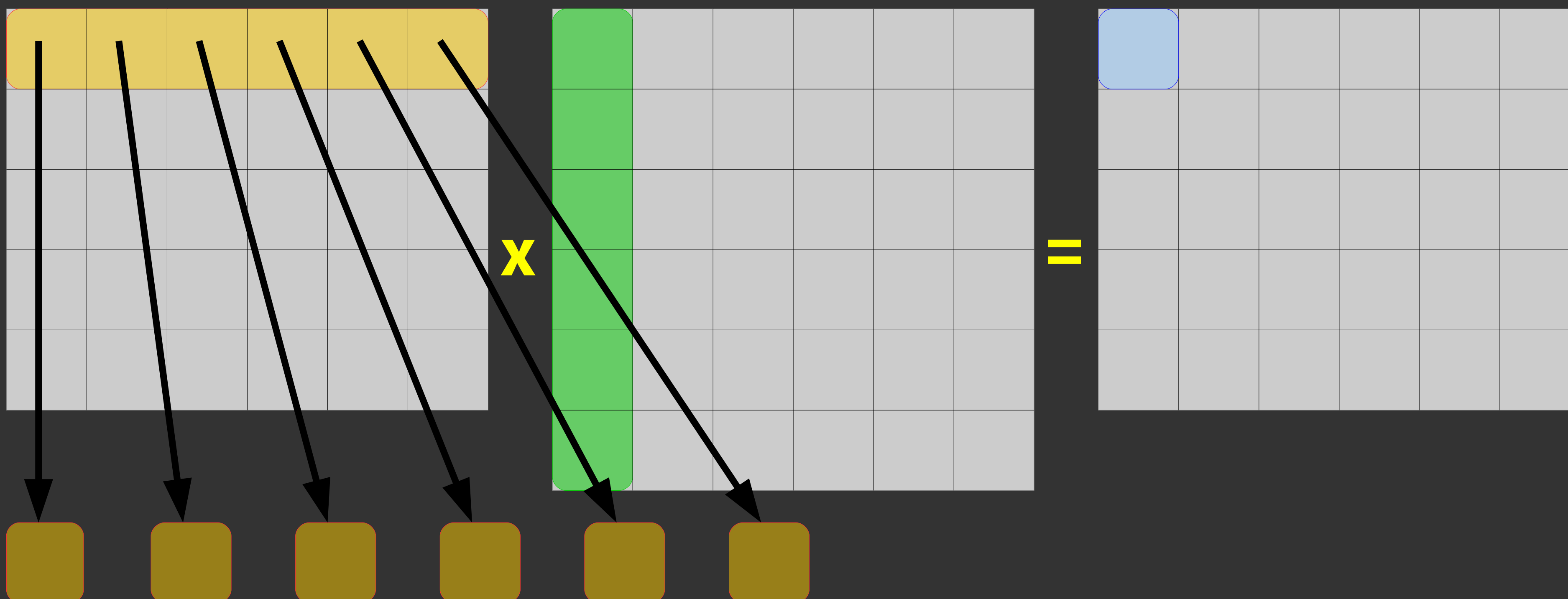
Standard Example: Matrix Multiplication



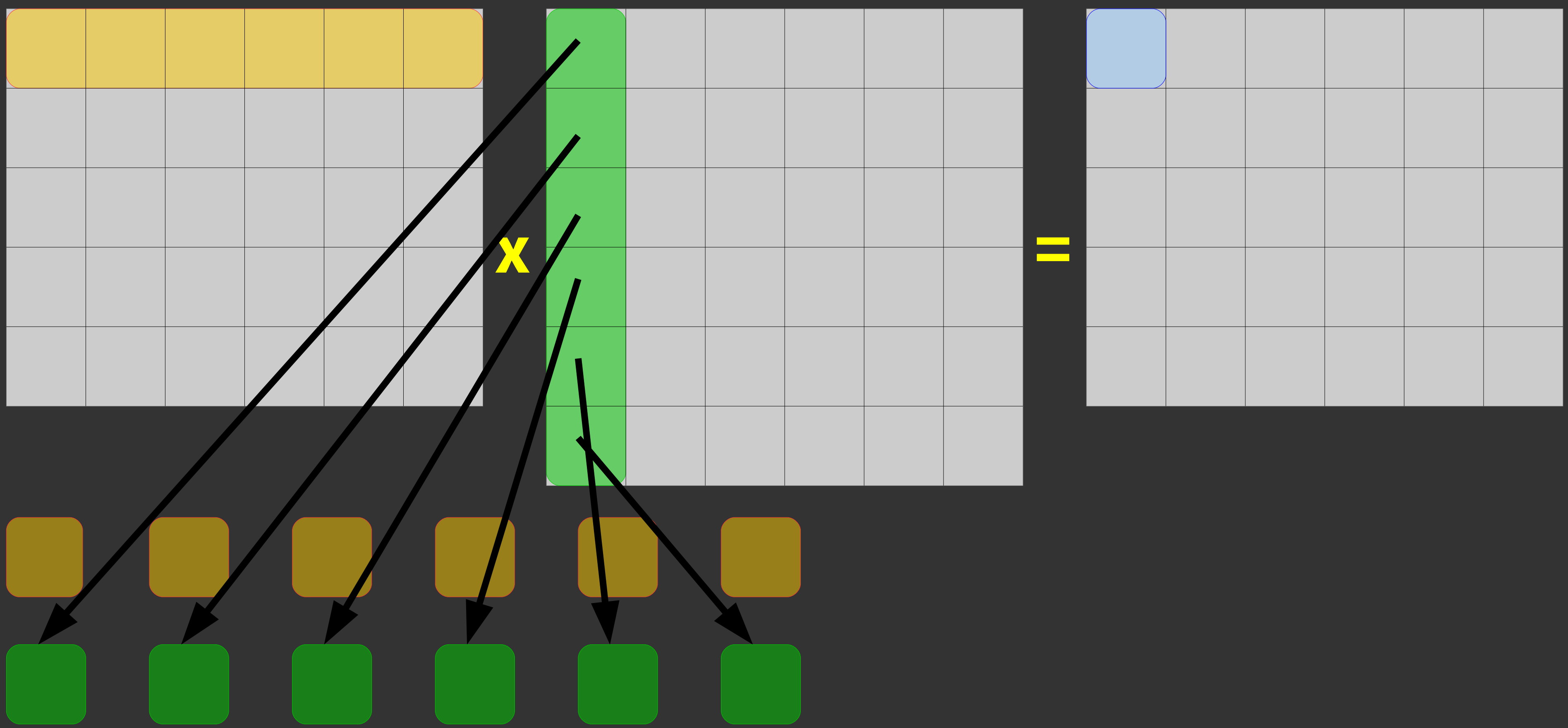
Standard Example: Matrix Multiplication



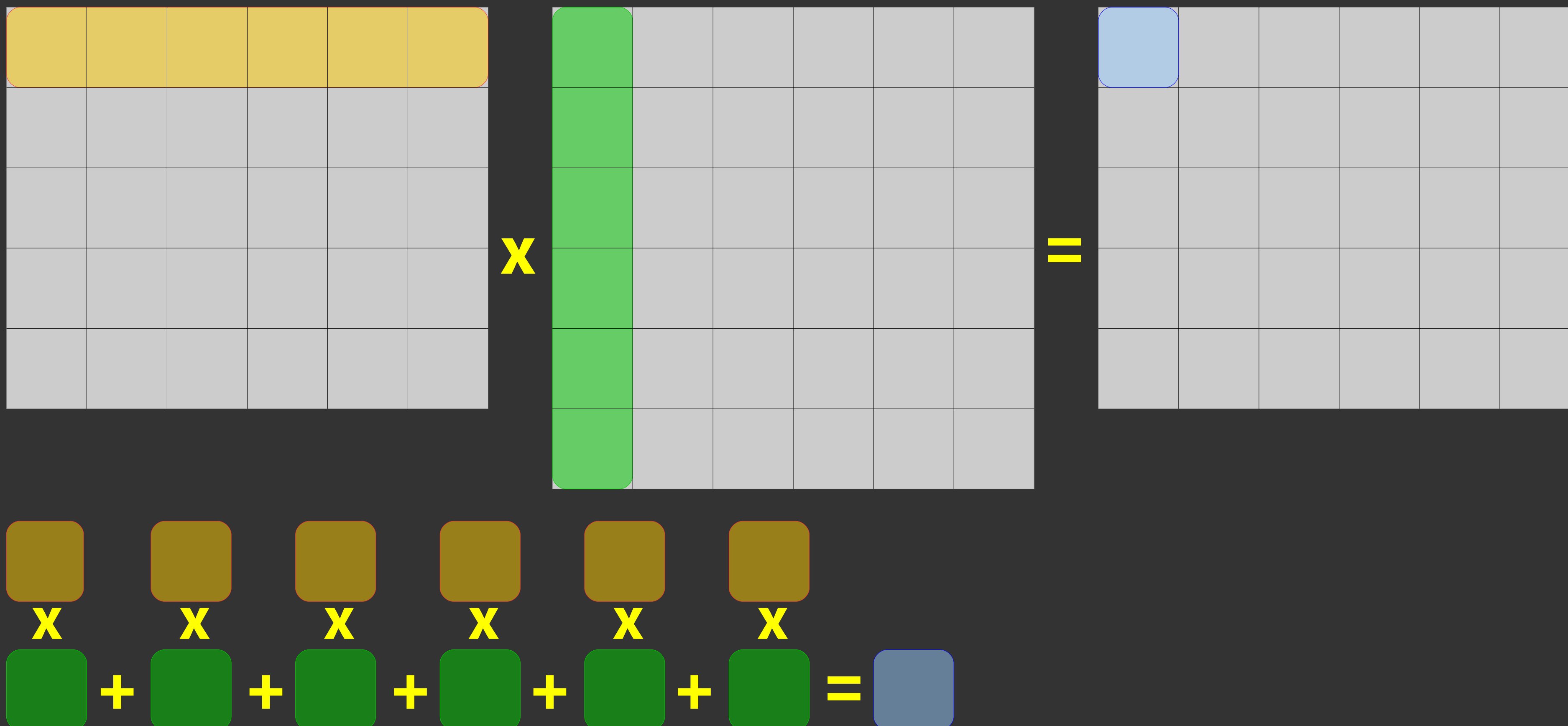
Standard Example: Matrix Multiplication



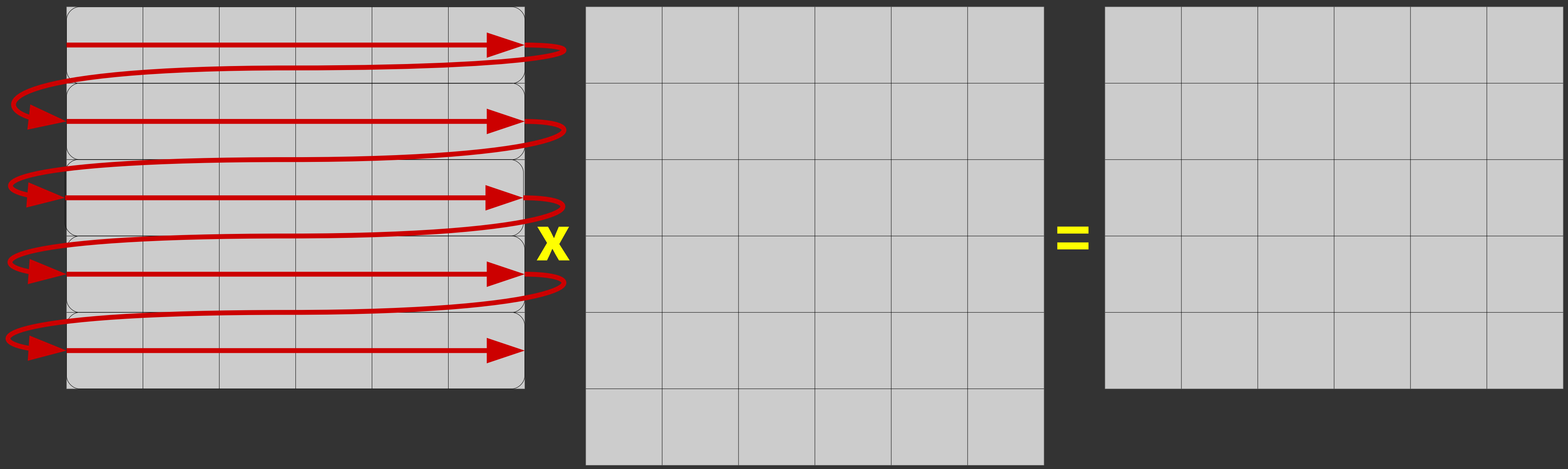
Standard Example: Matrix Multiplication



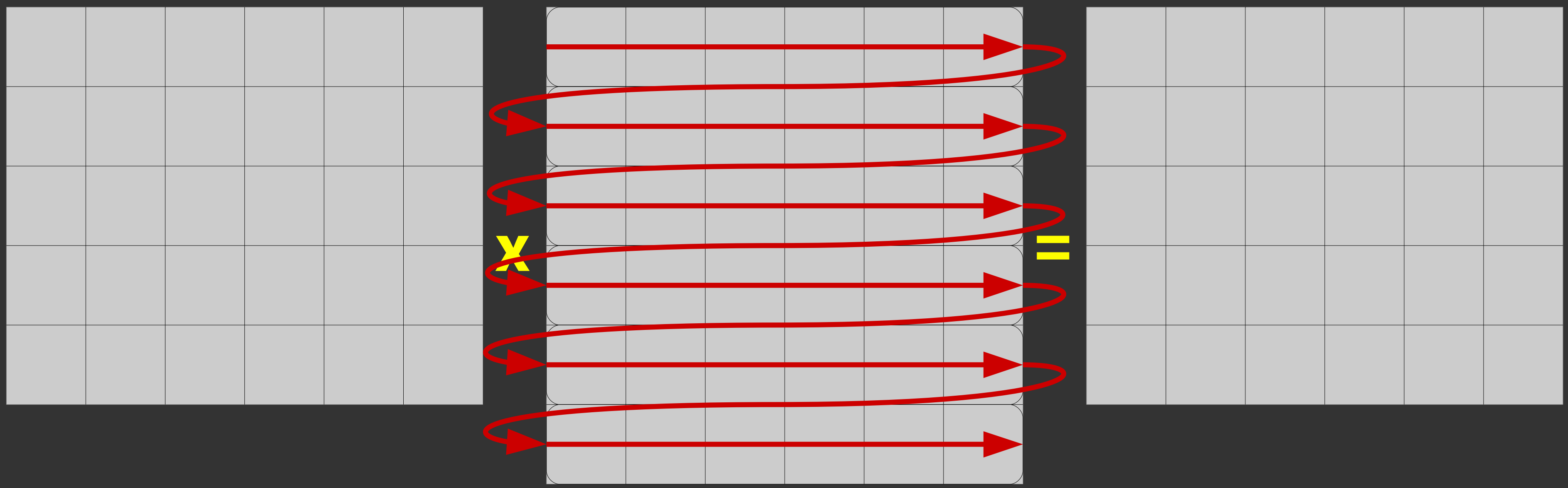
Standard Example: Matrix Multiplication



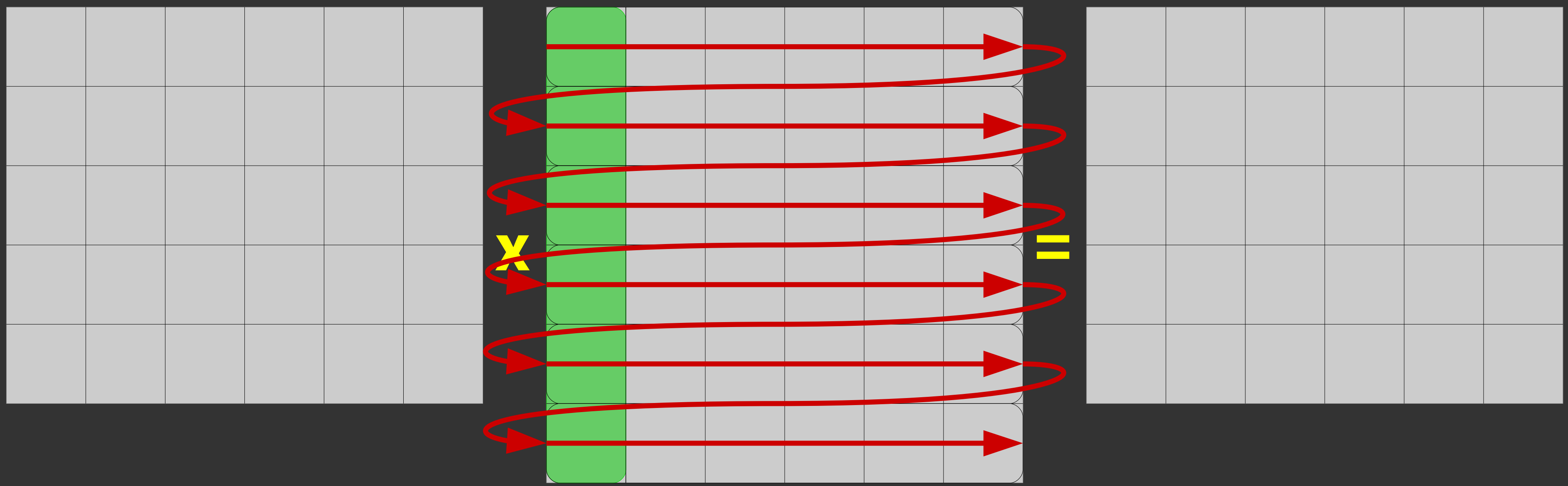
Memory Layout



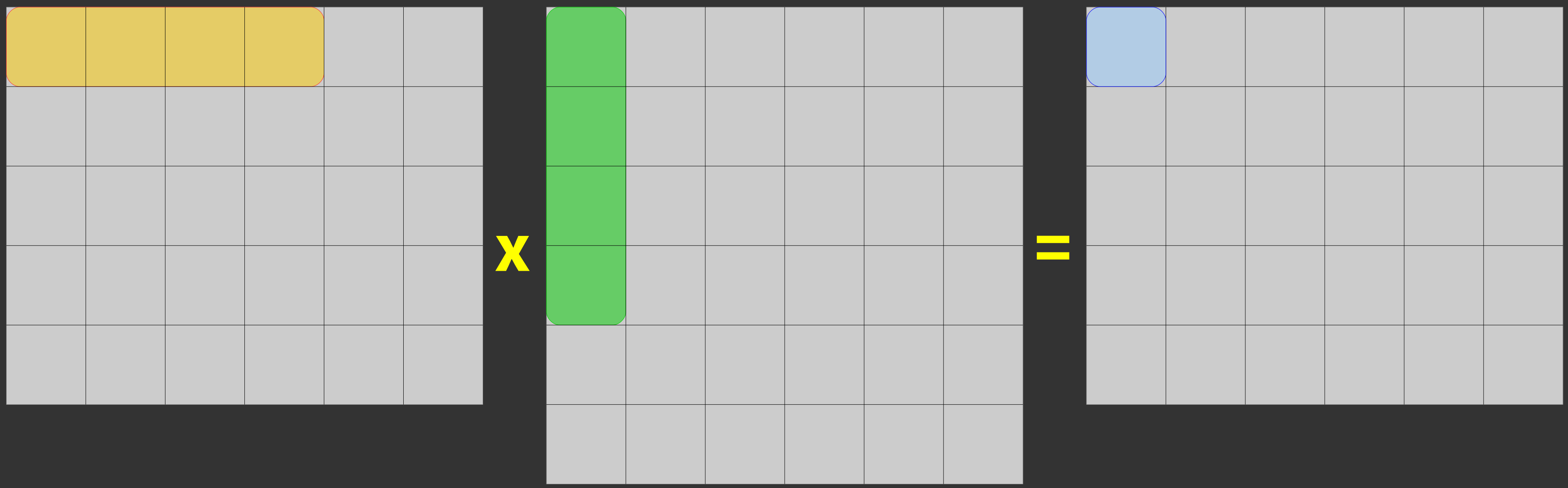
Memory Layout



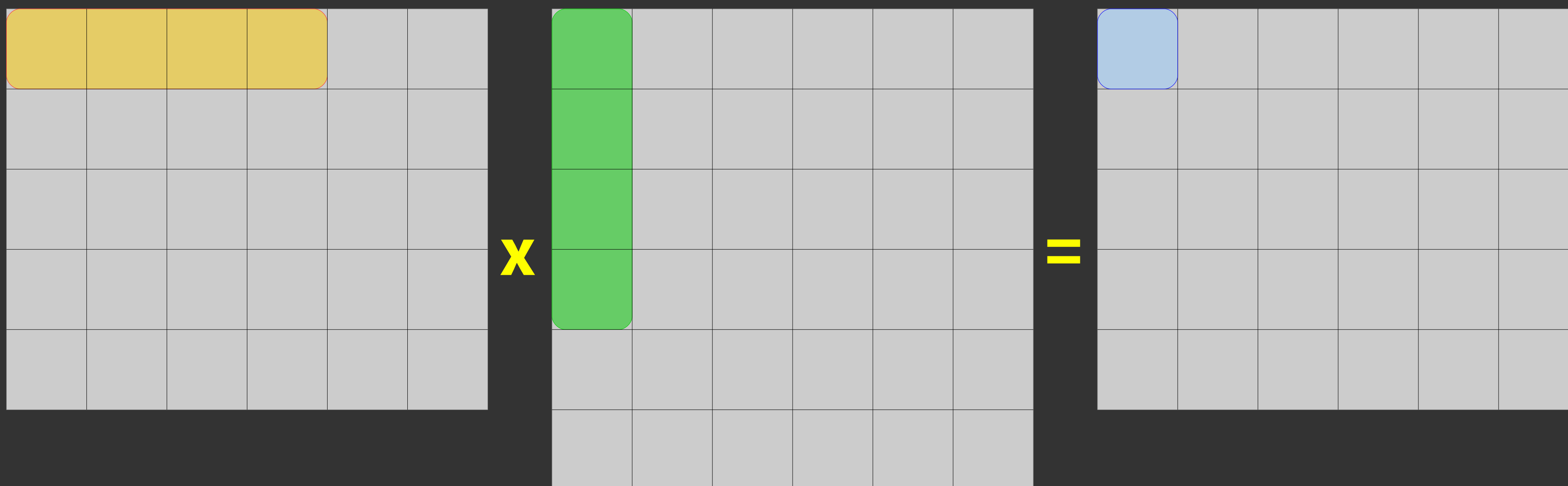
Memory Layout

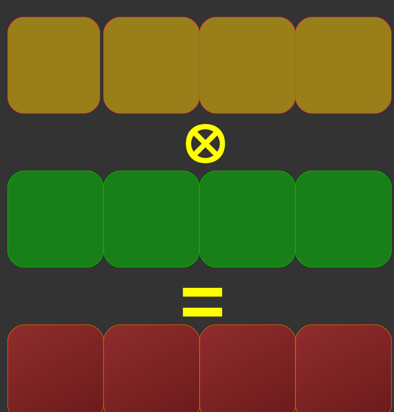


More Concrete: 4-element Vectors



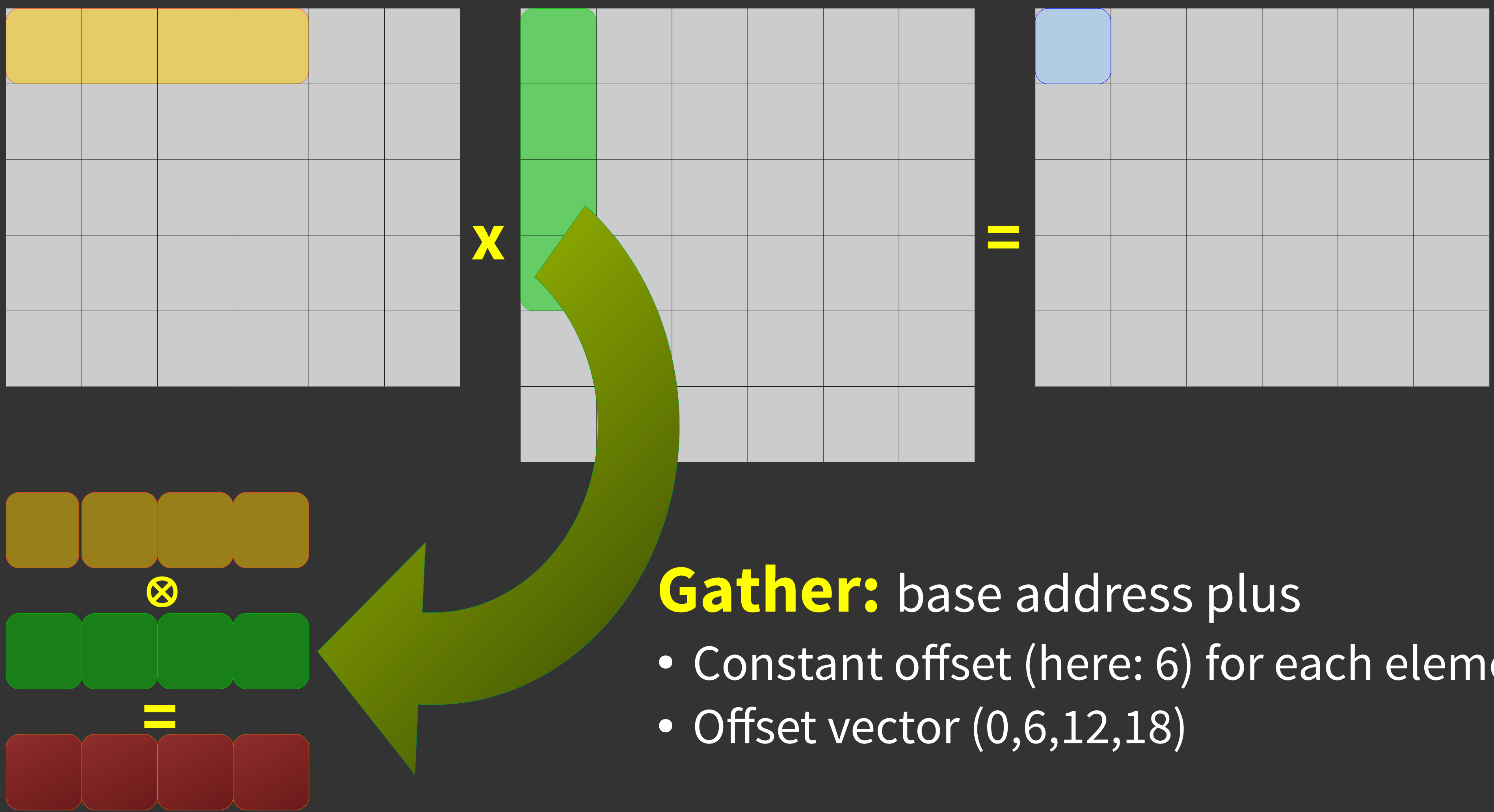
Vector Operations





⊗ **Element-wise multiplication**

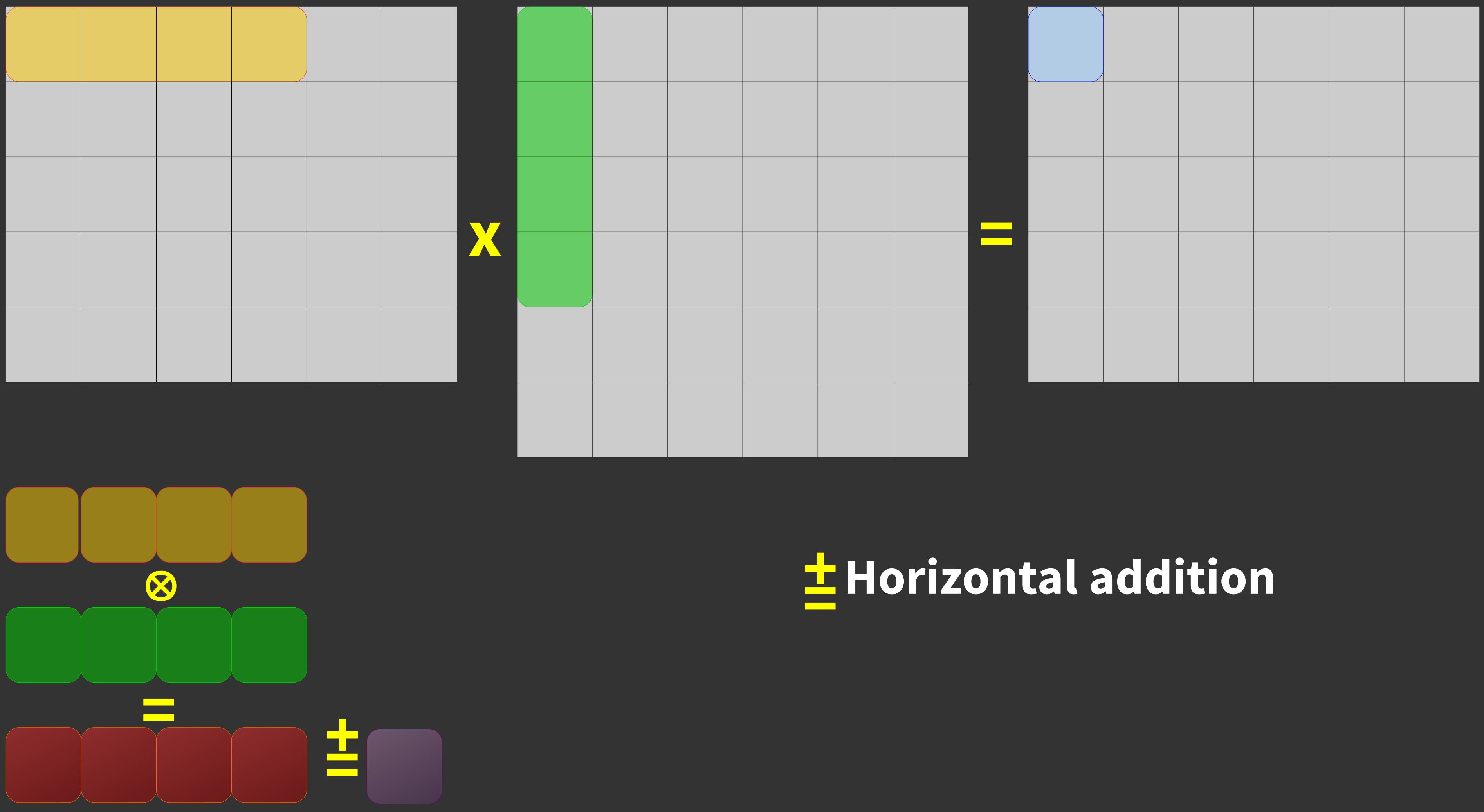
Non-Continuous Load



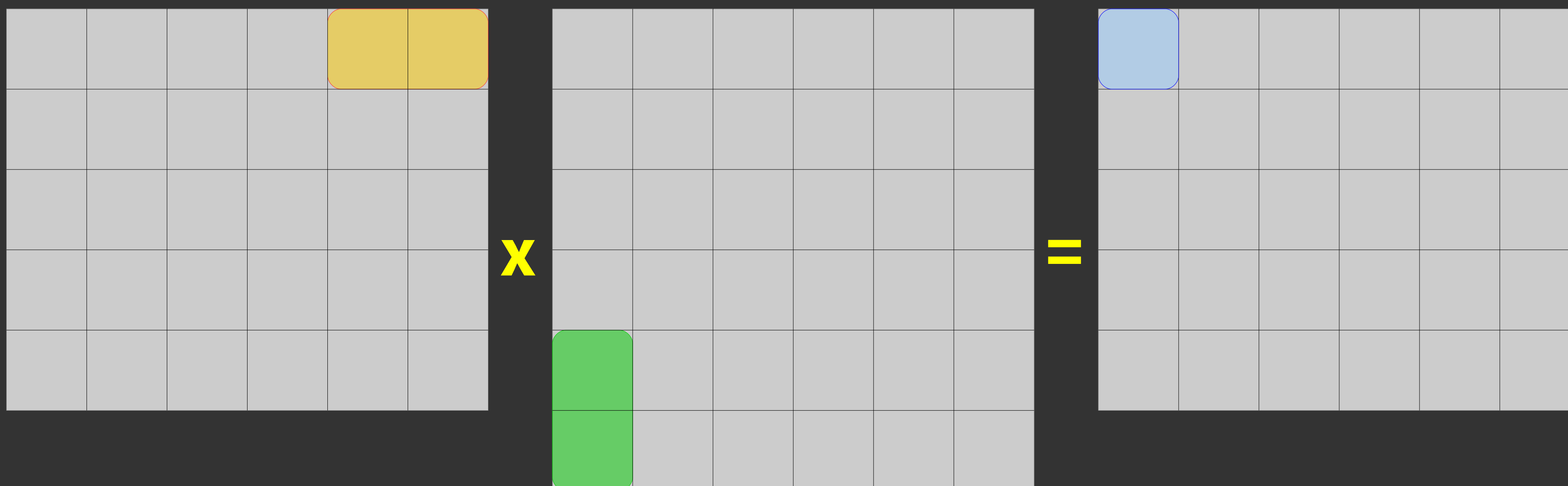
Gather: base address plus

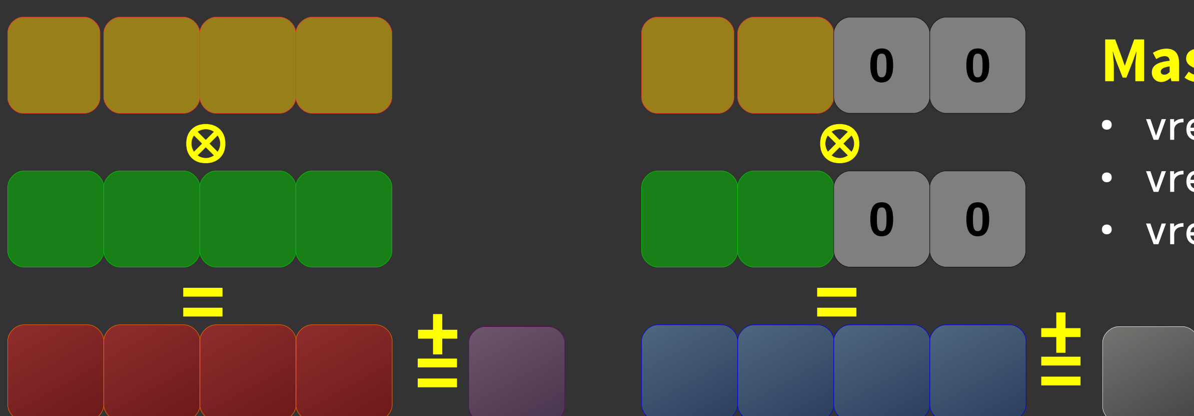
- Constant offset (here: 6) for each element
- Offset vector (0,6,12,18)

Intermediate Result



Selective Load

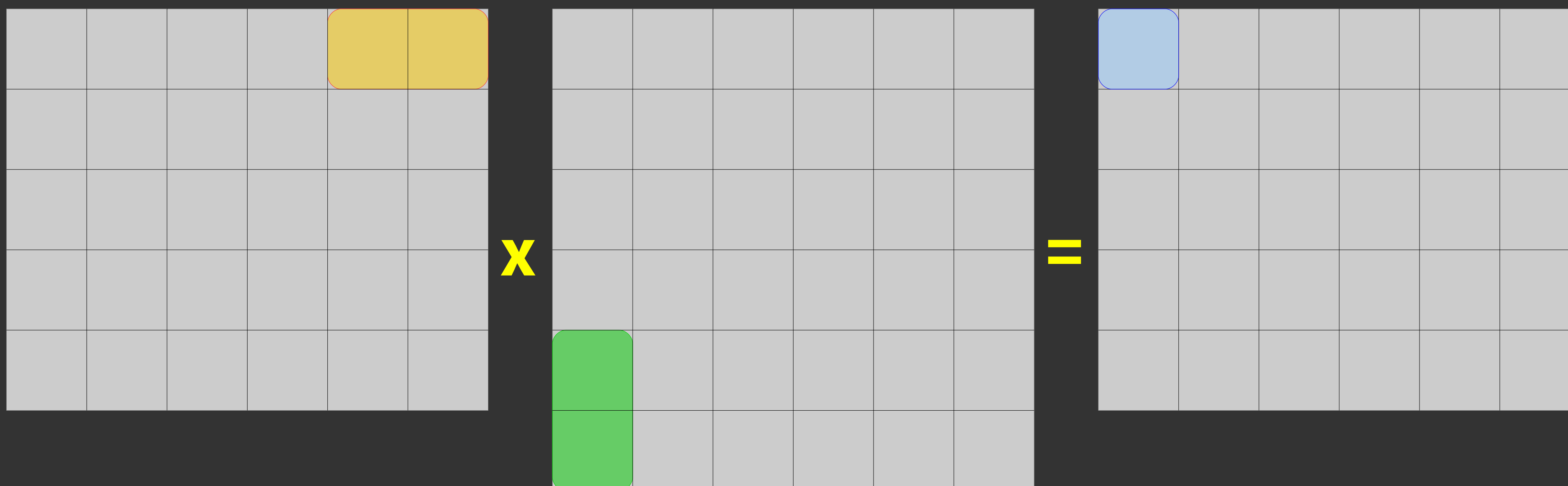


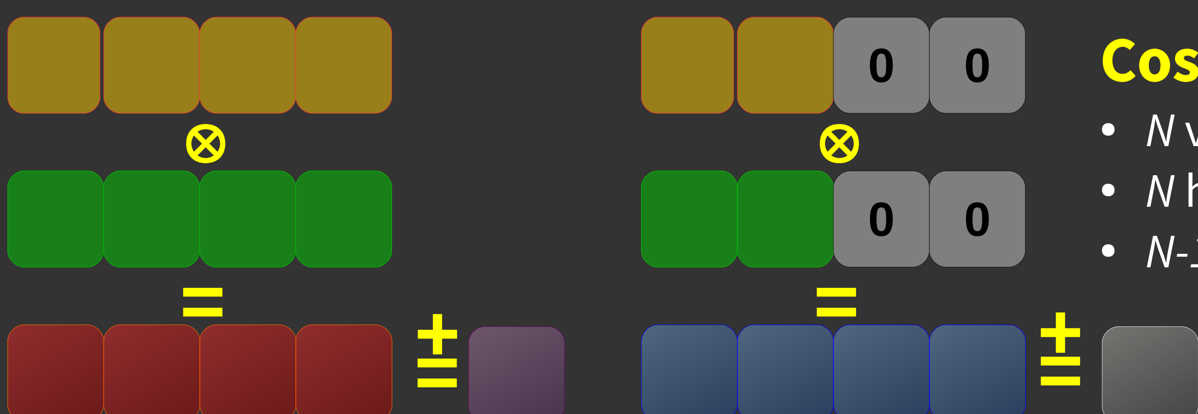


Masked operations:

- vreg1 = vloadz(src1, 0b1100)
- vreg2 = vgatherz(src2, 6, 0b1100)
- vreg3 = vaddz(vreg1, vreg2, vreg3, 0b1100)

Counting

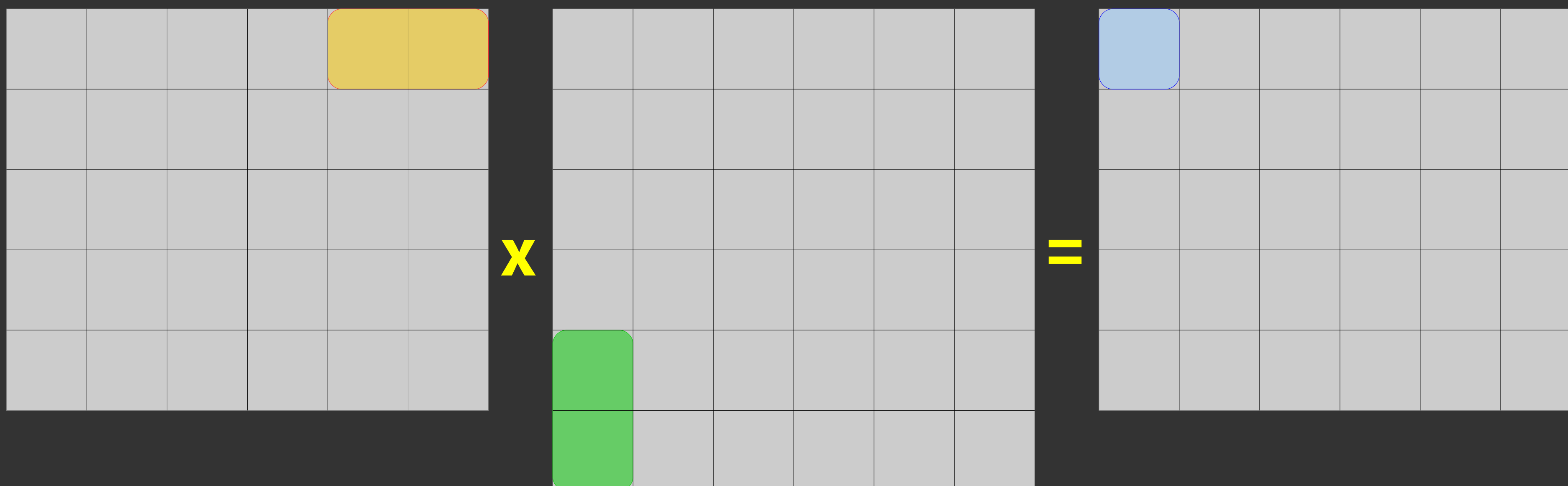


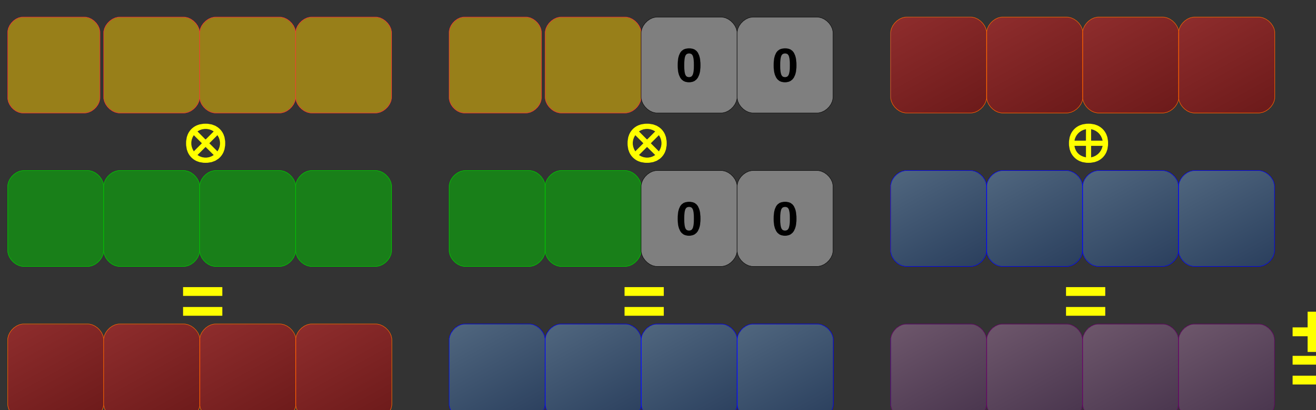


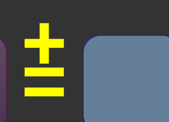
Cost:

- *N* vector multiplication
- *N* horizontal add (6 additions)
- *N-1* scalar additions

Addition is Associative!



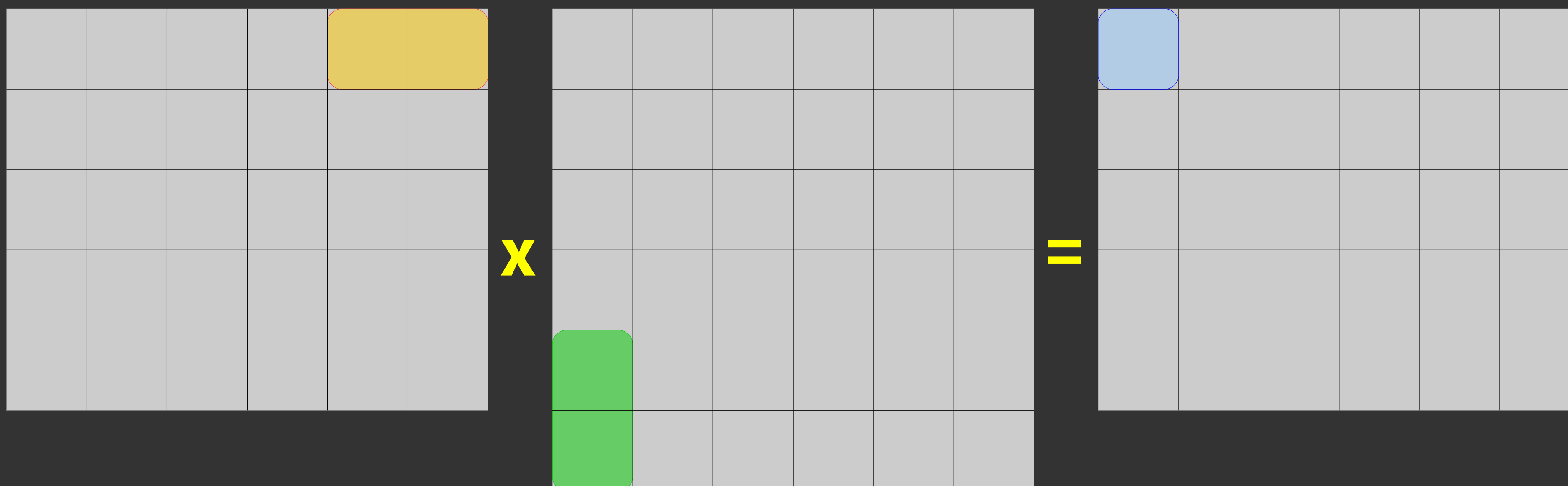


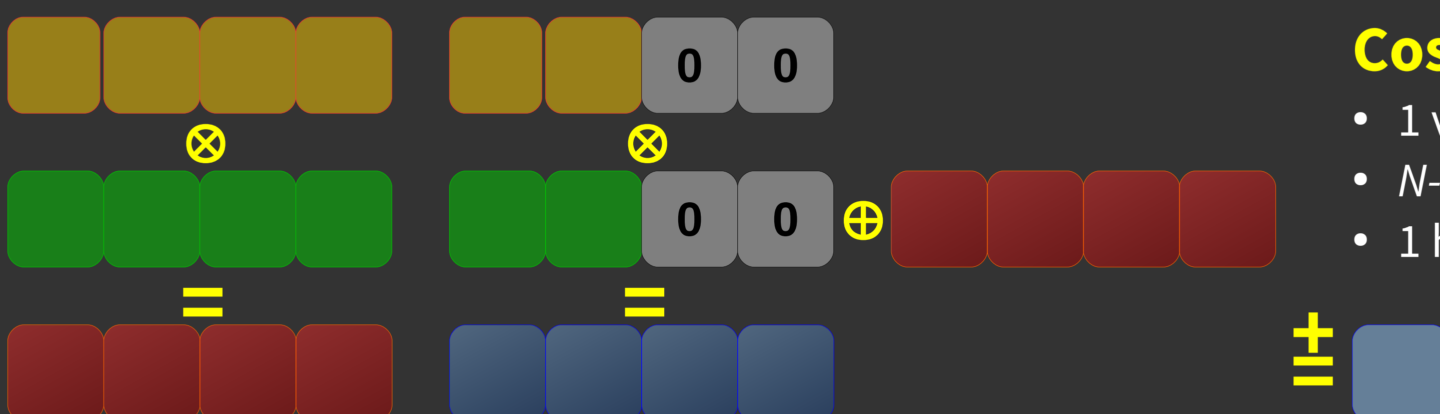


Cost:

- *N* vector multiplications
- *N-1* vector additions
- 1 horizontal add (3 additions)

Use FMA





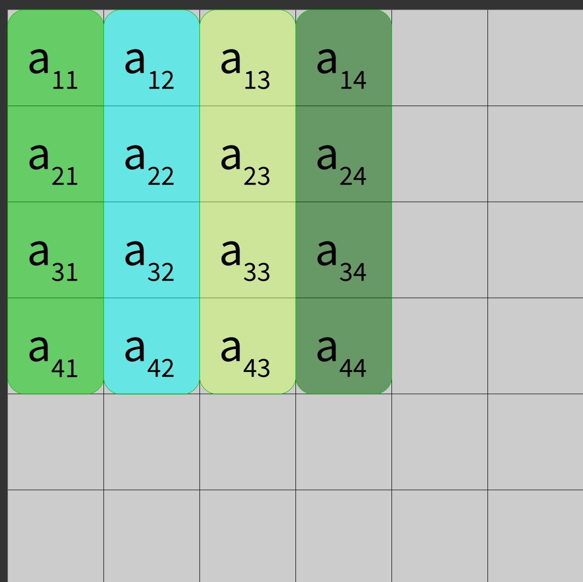
Cost:

- 1 vector multiplication
- *N-1* vector FMA
- 1 horizontal add (3 additions)

Used so far

- Gather/Scatter
- Masked load/store
- Masked arithmetic
- Horizontal addition
- FMA

More Optimal Load



Use continuous loads:

Transposition

Shuffle operations:

vreg1	a_{11}	a_{12}	a_{13}	a_{14}	vreg5	a_{11}	a_{21}	a_{12}	a_{22}	vreg5=unpacklo(vreg1,vreg2)
vreg2	a_{21}	a_{22}	a_{23}	a_{24}	vreg6	a_{31}	a_{41}	a_{32}	a_{42}	vreg6=unpacklo(vreg3,vreg4)
vreg3	a_{31}	a_{32}	a_{33}	a_{34}	vreg7	a_{13}	a_{23}	a_{14}	a_{24}	vreg7=unpackhi(vreg1,vreg2)
vreg4	a_{41}	a_{42}	a_{43}	a_{44}	vreg8	a_{33}	a_{43}	a_{34}	a_{44}	vreg8=unpackhi(vreg3,vreg4)

Transposition II

More shuffle operations:

vreg5	a_{11}	a_{21}	a_{12}	a_{22}	vreg1	a_{11}	a_{21}	a_{31}	a_{41}	vreg1=movlh(vreg5,vreg6)
vreg6	a_{31}	a_{41}	a_{32}	a_{42}	vreg2	a_{12}	a_{22}	a_{32}	a_{42}	vreg2=movhl(vreg6,vreg5)
vreg7	a_{13}	a_{23}	a_{14}	a_{24}	vreg3	a_{13}	a_{23}	a_{33}	a_{43}	vreg3=movlh(vreg7,vreg8)
vreg8	a_{33}	a_{43}	a_{34}	a_{44}	vreg4	a_{14}	a_{24}	a_{34}	a_{44}	vreg4=movhl(vreg8,vreg7)

Available Code

- No need to reinvent the wheel:

  BLAS, LAPACK, …
- Different implementations
  - Netlib
  - ATLAS (Automatically Tuned Linear Algebra System)
  - Eigen (C++ template classes)

Create Vector Version

```
double exp(double x) {
  di_t j1 = { .x = x };
  int m = j1.i[HIGH_HALF];
  int n = m & hugeint;
  if (n > smallint && n < badint) {
    double y = x * log2e + three51;
    double bexp = y - three51;
    j1.x = y;
    double eps = bexp * ln_two2;
    double t = bexp * -ln_two1 + x;
    y = t + three33;
    double base = y - three33;
    di_t j2 = { .x = y };
    double del = (t - base) - eps;
    eps = (p3 * del + p2)*del*del + del;
    int i = ((j2.i[LOW_HALF] >> 8)
             & 0xfffffffe) + 356;
    int j = (j2.i[LOW_HALF] & 511)<<1;
    double al = coar[i] * fine[j];
    double bet
      = (coar[i+1] * fine[j+1]
         + coar[i] * fine[j+1]
         + coar[i + 1] * fine[j]);
double rem = al * eps + bet * eps + bet;
double res = al + rem;
double cor = (al - res) + rem;
int ex = j1.i[LOW_HALF];
di_t binexp = {{0, 0}};
if (n < bigint)  {
test_mult_return:
  if (res == cor * err_0 + res) {
    binexp.i[HIGH_HALF] = (ex+1023)<<20;
    return res * binexp.x;
  }
} else if ((m & 0x80000000) == 0) {
  if (res == cor * err_0 + res) {
    binexp.i[HIGH_HALF] = (ex+767)<< 20;
    return res * binexp.x * t256;
  }
} else {
  if (res < 1.0) {
    res += res; cor += cor; ex -= 1;
  }
  if (ex >= DBL_MIN_EXP - 1)
    goto test_mult_return;
```

Problems Converting

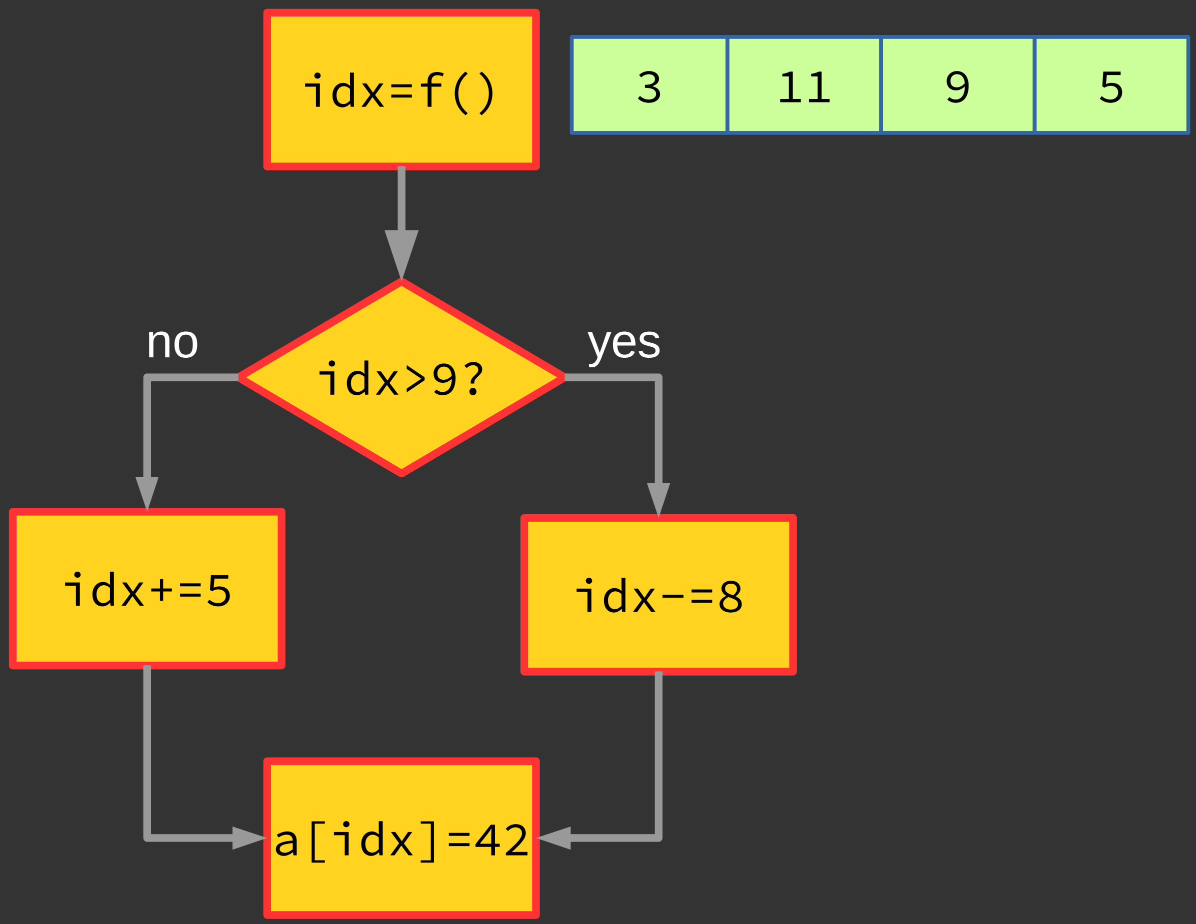
```
double exp(double x) {
  di_t j1 = { .x = x };
  int m = j1.i[HIGH_HALF];
  int n = m & hugeint;
  if (n > smallint && n < badint) {
    double y = x * log2e + three51;
    double bexp = y - three51;
    j1.x = y;
    double eps = bexp * ln_two2;
    double t = bexp * -ln_two1 + x;
    y = t + three33;
    double base = y - three33;
    di_t j2 = { .x = y };
    double del = (t - base) - eps;
    eps = (p3 * del + p2)*del*del + del;
    int i = ((j2.i[LOW_HALF] >> 8)
             & 0xfffffffe) + 356;
    int j = (j2.i[LOW_HALF] & 511)<<1;
    double al = coar[i] * fine[j];
    double bet
      = (coar[i+1] * fine[j+1]
         + coar[i] * fine[j+1]
         + coar[i + 1] * fine[j]);
```

```
    double rem = al * eps + bet * eps + bet;
    double res = al + rem;
    double cor = (al - res) + rem;
    int ex = j1.i[LOW_HALF];
    di_t binexp = {{0, 0}};
    if (n < bigint) {
    test_mult_return:
      if (res == cor * err_0 + res) {
        binexp.i[HIGH_HALF] = (ex+1023)<<20;
        return res * binexp.x;
      }
    } else if ((m & 0x80000000) == 0) {
      if (res == cor * err_0 + res) {
        binexp.i[HIGH_HALF] = (ex+767)<< 20;
        return res * binexp.x * t256;
      }
    } else {
      if (res < 1.0) {
        res += res; cor += cor; ex -= 1;
      }
      if (ex >= DBL_MIN_EXP - 1)
        goto test_mult_return;
```

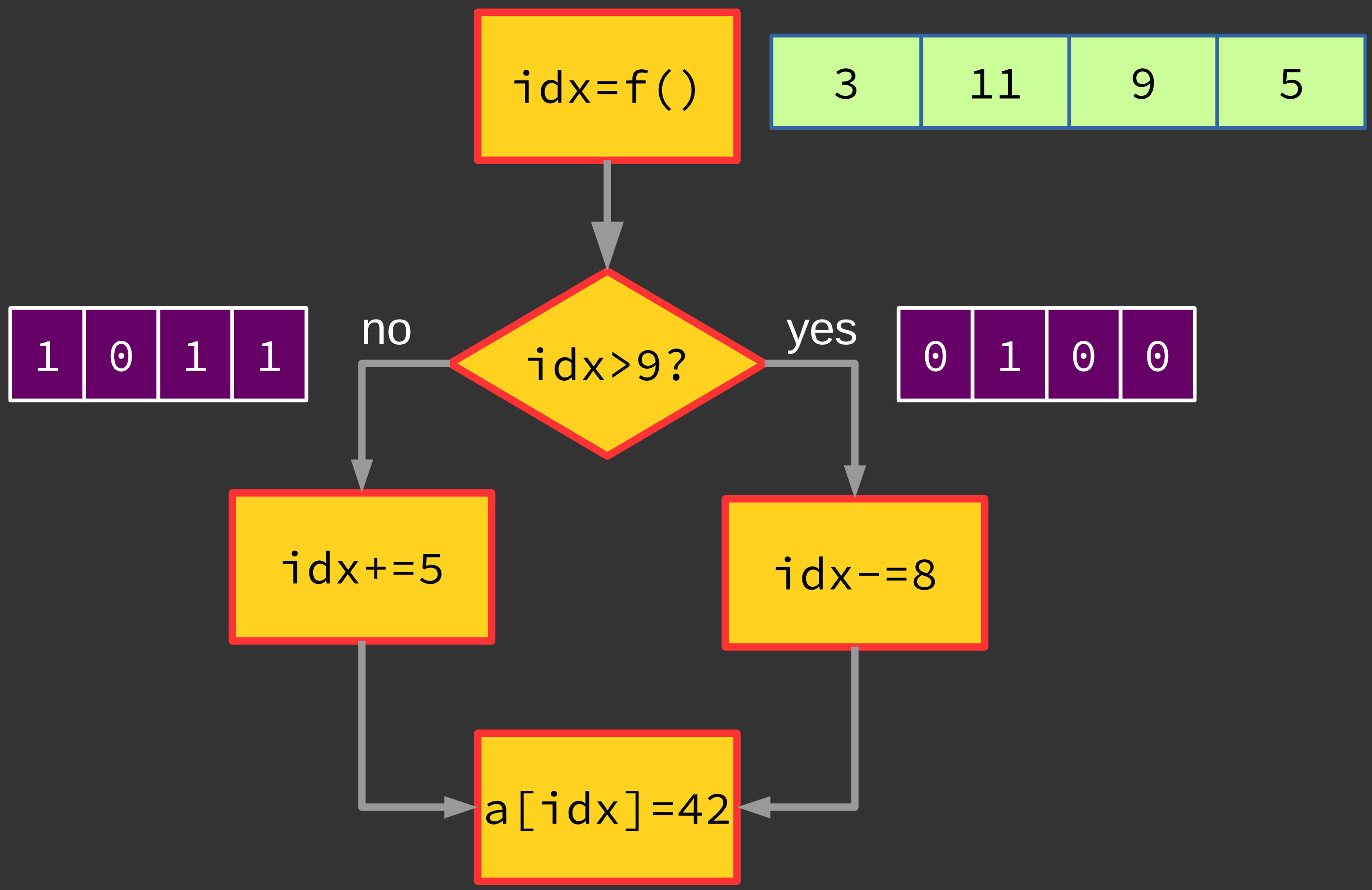
if : Bad!

Linear Code: Good!

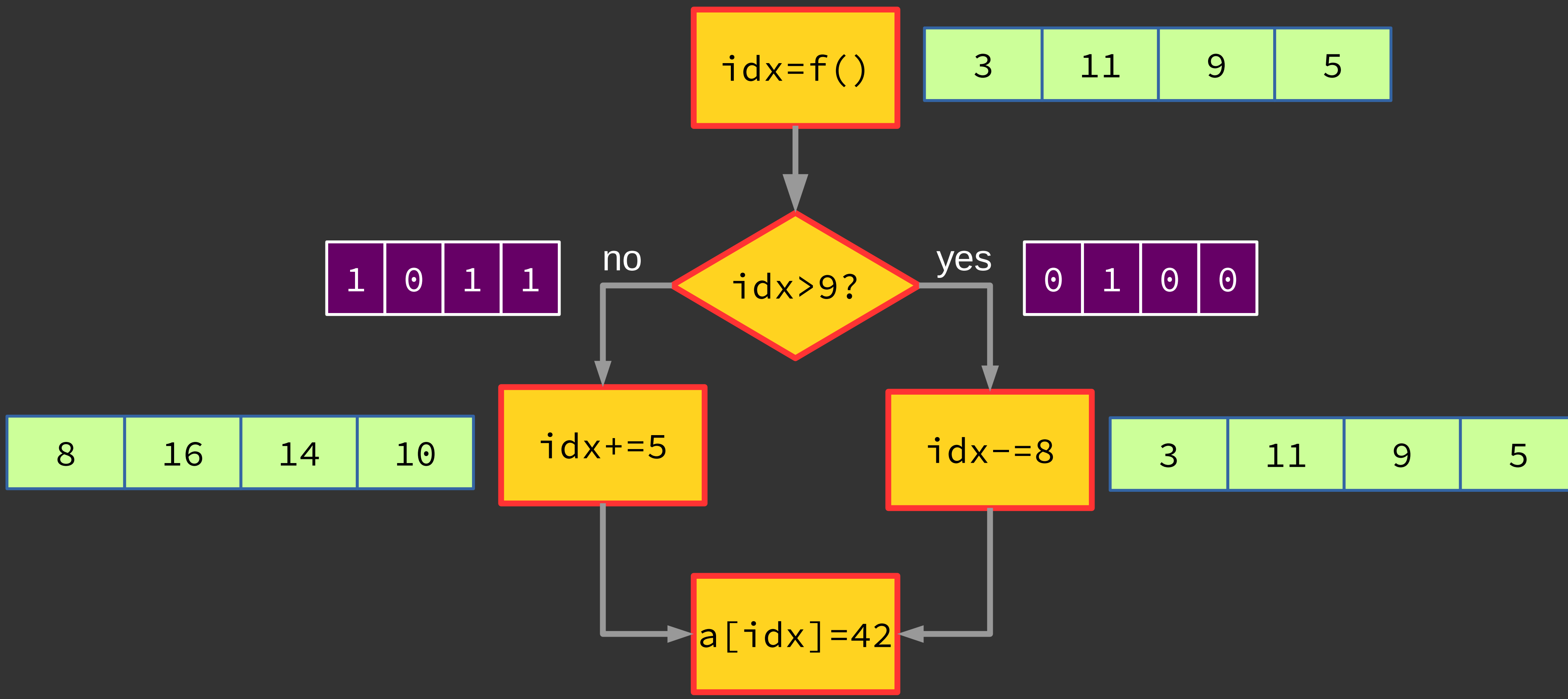
Conditional Code



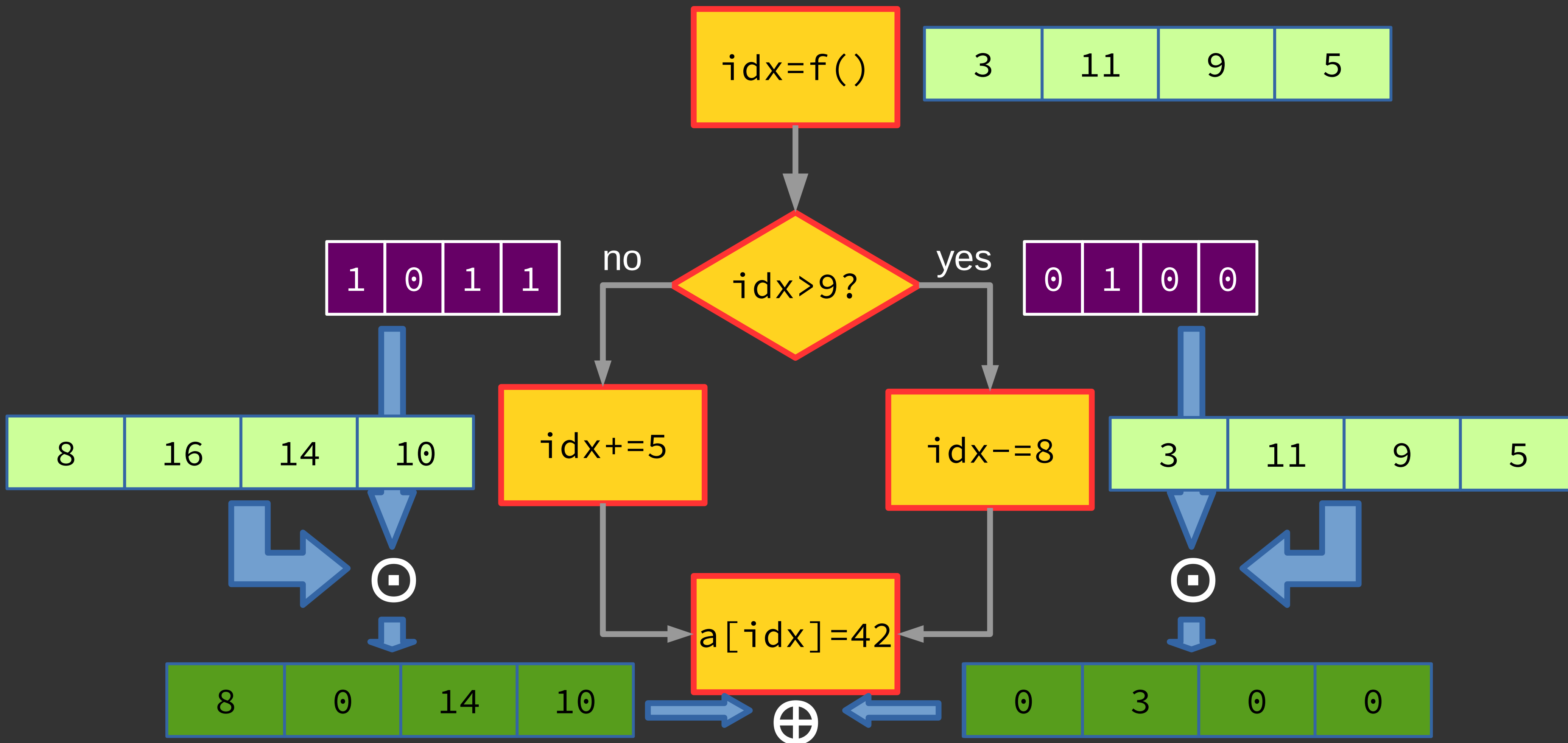
Compute Bitmasks



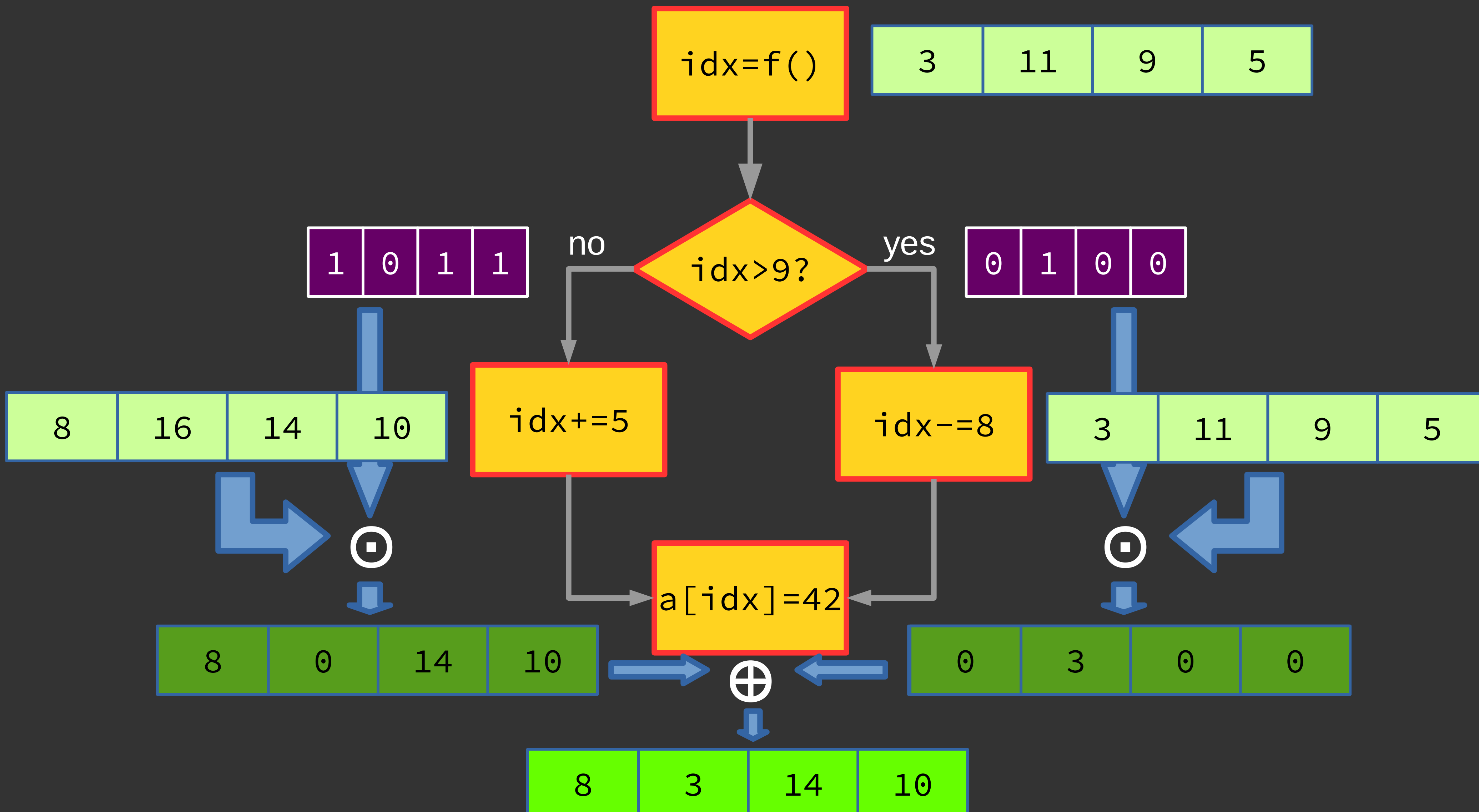
Unconditionally Compute



Compute Partial Results



Final Result



Instruction Sequence

```
idx = f();
mask = vcmp(idx, broadcast(9), CMP_GT);
tmp_t = vadd(idx, broadcast(5));
tmp_f = vsub(idx, broadcast(8));
idx = vor(vand(tmp_t, mask),
          vnand(tmp_f, mask));
vscatter(a, idx, broadcast(42));
```

Special Merge Instruction

```
idx = f();
mask = vcmp(idx, broadcast(9), CMP_GT);
tmp_t = vadd(idx, broadcast(5));
tmp_f = vsub(idx, broadcast(8));
idx = vblend(tmp_t, tmp_f, mask);

vscatter(a, idx, broadcast(42));
```

Compiler Help

- Closer to not need using assembler code or compiler intrinsics
  - OpenMP 4.0: `#pragma omp simd`
  - OpenACC 2.0: `#pragma acc parallel loop`
  - Or just normal optimization

Simple Code Sequence

```
void fct(double *r,
         const double *a,
         const double *b,
         double f)
{
  for (unsigned i = 0; i < 128; ++i)
    r[i] = a[i] * f + b[i];
}
```

Compile with

gcc -c -O3 -march=haswell fct.cc

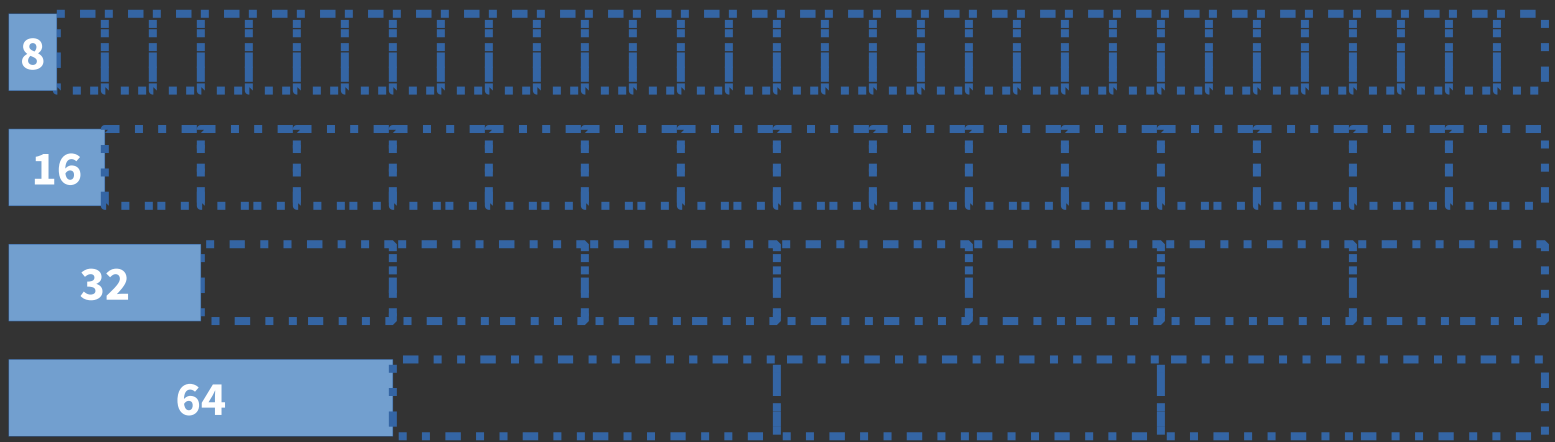
Result

```
0000000000000000 <fct(double*, double const*, double const*, double)>:
   0:   48 8d 4a 20             lea    0x20(%rdx),%rcx
   4:   48 8d 47 20             lea    0x20(%rdi),%rax
   8:   48 39 cf                cmp    %rcx,%rdi
[…]
  e0:   83 c1 01                add    $0x1,%ecx
  e3:   c4 c1 7d 28 0c 03       vmovapd (%r11,%rax,1),%ymm1
  e9:   c4 c2 ed a8 0c 02       vfmadd213pd (%r10,%rax,1),%ymm2,%ymm1
  ef:   c4 c1 7d 11 0c 00       vmovupd %ymm1,(%r8,%rax,1)
  f5:   48 83 c0 20             add    $0x20,%rax
  f9:   44 39 c9                cmp    %r9d,%ecx
  fc:   72 e2                   jb     e0 <fct+0xe0>
[…]
 1e3:   bb 01 00 00 00          mov    $0x1,%ebx
 1e8:   e9 c3 fe ff ff          jmpq   b0 <fct+0xb0>
```

Alignment

```
typedef double double_32 attribute((aligned(32)));
void fct(double_32 *r,
         const double_32 *a,
         const double_32 *b,
         double f)
{
  for (unsigned i = 0; i < 128; ++i)
    r[i] = a[i] * f + b[i];
}
```

Vector Register Use



Integer: 8-, 16-, 32-, and 64-bit: **97%, 94%, 88%, and 75% unused**

FP: 16-, 32-, and 64-bit: **94%, 88%, and 75% unused**

Much Better

```
0000000000000000 <fct(double*, double const*, double
const*, double)>:
   0: 48 8d 4a 20             lea    0x20(%rdx),%rcx
   4: 48 8d 47 20             lea    0x20(%rdi),%rax
   8: 48 39 cf                cmp    %rcx,%rdi
   b: 41 0f 93 c0             setae  %r8b
   f: 48 39 c2                cmp    %rax,%rdx
  12: 0f 93 c1                setae  %cl
  15: 41 08 c8                or     %cl,%r8b
  18: 74 46                   je     60 <fct+0x60>
  1a: 48 8d 4e 20             lea    0x20(%rsi),%rcx
  1e: 48 39 cf                cmp    %rcx,%rdi
  21: 0f 93 c1                setae  %cl
  24: 48 39 c6                cmp    %rax,%rsi
  27: 0f 93 c0                setae  %al
  2a: 08 c1                   or     %al,%cl
  2c: 74 32                   je     60 <fct+0x60>
  2e: c4 e2 7d 19 c0          vbroadcastsd %xmm0,%ymm0
  33: 31 c0                   xor    %eax,%eax
  35: 0f 1f 00                nopl   (%rax)
```

Vectorized Loop

```
  38: c5 fd 28 0c 06          vmovapd (%rsi,%rax,1),%ymm1
  3d: c4 e2 fd a8 0c 02       vfmadd213pd (%rdx,%rax,1),%ymm0,%ymm1
  43: c5 fd 29 0c 07          vmovapd %ymm1,(%rdi,%rax,1)
  48: 48 83 c0 20             add    $0x20,%rax
  4c: 48 3d 00 04 00 00       cmp    $0x400,%rax
  52: 75 e4                   jne    38 <fct+0x38>
  54: c5 f8 77                vzeroupper
  57: c3                      retq
  58: 0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
  5f: 00
  60: 31 c0                   xor    %eax,%eax
  62: 66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
  68: c5 fb 10 0c 06          vmovsd (%rsi,%rax,1),%xmm1
  6d: c4 e2 f9 a9 0c 02       vfmadd213sd (%rdx,%rax,1),%xmm0,%xmm1
  73: c5 fb 11 0c 07          vmovsd %xmm1,(%rdi,%rax,1)
  78: 48 83 c0 08             add    $0x8,%rax
  7c: 48 3d 00 04 00 00       cmp    $0x400,%rax
  82: 75 e4                   jne    68 <fct+0x68>
  84: c3                      retq
```

Still Scalar Loop

Available Since ISO C99

```
typedef double double_32 attribute((aligned(32)));
void fct(double_32 *__restrict r,
         const double_32 *__restrict a,
         const double_32 *__restrict b,
         double f)
{
  for (unsigned i = 0; i < 128; ++i)
    r[i] = a[i] * f + b[i];
}
```

Perfect!

```
0000000000000000 <fct(double*, double const*, double const*, double)>:
   0: c4 e2 7d 19 c0          vbroadcastsd %xmm0,%ymm0
   5: 31 c0                   xor    %eax,%eax
   7: 66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
   e: 00 00
  10: c5 fd 28 0c 06          vmovapd (%rsi,%rax,1),%ymm1
  15: c4 e2 fd a8 0c 02       vfmadd213pd (%rdx,%rax,1),%ymm0,%ymm1
  1b: c5 fd 29 0c 07          vmovapd %ymm1,(%rdi,%rax,1)
  20: 48 83 c0 20             add    $0x20,%rax
  24: 48 3d 00 04 00 00       cmp    $0x400,%rax
  2a: 75 e4                   jne    10 <fct+0x10>
  2c: c5 f8 77                vzeroupper
  2f: c3                      retq
```

OpenMP is simpler

```
void fct(double *r,
         const double *a,
         const double *b,
         double f)
{
#pragma omp simd aligned(r,a,b:32)
  for (unsigned i = 0; i < 128; ++i)
    r[i] = a[i] * f + b[i];
}
```

Architecture Dependence

```
__attribute__((__target__("default")))
void fct(double *__restrict r, const double *__restrict a, const double *__restrict b, double f) {
#pragma omp simd aligned(r,a,b:32)
  for (unsigned i = 0; i < 128; ++i)
    r[i] = a[i] * f + b[i];
}

__attribute__((__target__("avx")))
void fct(double *__restrict r, const double *__restrict a, const double *__restrict b, double f) {
#pragma omp simd aligned(r,a,b:32)
  for (unsigned i = 0; i < 128; ++i)
    r[i] = a[i] * f + b[i];
}

__attribute__((__target__("arch=haswell")))
void fct(double *__restrict r, const double *__restrict a, const double *__restrict b, double f) {
#pragma omp simd aligned(r,a,b:32)
  for (unsigned i = 0; i < 128; ++i)
    r[i] = a[i] * f + b[i];
}
```

Architecture Dependence

```
__attribute__((__target__("default")))
void fct(double *__restrict r, const double *__restrict a, const double *__restrict b, double f) {
#pragma omp simd aligned(r,a,b:32)
  for (unsigned i = 0; i < 128; ++i)
    r[i] = a[i] * f + b[i];
}

__attribute__((__target__("avx")))
void fct(double *__restrict r, const double *__restrict a, const double *__restrict b, double f) {
#pragma omp simd aligned(r,a,b:32)
  for (unsigned i = 0; i < 128; ++i)
    r[i] = a[i] * f + b[i];
}

__attribute__((__target__("arch=haswell")))
void fct(double *__restrict r, const double *__restrict a, const double *__restrict b, double f) {
#pragma omp simd aligned(r,a,b:32)
  for (unsigned i = 0; i < 128; ++i)
    r[i] = a[i] * f + b[i];
}
```

Multi-Version Code

```
0000000000000000 <_Z3fctPdPKdS1_d>:
   0: xor    %eax,%eax
   2: unpcklpd %xmm0,%xmm0
   6: nopw   %cs:0x0(%rax,%rax,1)
  10: movapd (%rsi,%rax,1),%xmm1
  15: mulpd  %xmm0,%xmm1
  19: addpd  (%rdx,%rax,1),%xmm1
  1e: movaps %xmm1,(%rdi,%rax,1)
  22: add    $0x10,%rax
  26: cmp    $0x400,%rax
  2c: jne    10 <_Z3fctPdPKdS1_d+0x10>
  2e: retq
  2f: nop
```

```
0000000000000030 <_Z3fctPdPKdS1_d.avx>:
  30: xor    %eax,%eax
  32: vmovddup %xmm0,%xmm1
  36: vinsertf128 $0x1,%xmm1,%ymm1,%ymm1
  3c: nopl   0x0(%rax)
  40: vmulpd (%rsi,%rax,1),%ymm1,%ymm0
  45: vaddpd (%rdx,%rax,1),%ymm0,%ymm0
  4a: vmovapd %ymm0,(%rdi,%rax,1)
  4f: add    $0x20,%rax
  53: cmp    $0x400,%rax
  59: jne    40 <_Z3fctPdPKdS1_d.avx+0x10>
  5b: vzeroupper
  5e: retq
  5f: nop
```

```
0000000000000060 <_Z3fctPdPKdS1_d.arch_haswell>:
  60: vbroadcastsd %xmm0,%ymm0
  65: xor    %eax,%eax
  67: nopw   0x0(%rax,%rax,1)
  70: vmovapd (%rsi,%rax,1),%ymm1
  75: vfmadd213pd (%rdx,%rax,1),%ymm0,%ymm1
  7b: vmovapd %ymm1,(%rdi,%rax,1)
  80: add    $0x20,%rax
  84: cmp    $0x400,%rax
  8a: jne    70 <_Z3fctPdPKdS1_d.arch_haswell+0x10>
  8c: vzeroupper
  8f: retq
```

Vector Types

```
typedef double vdouble __attribute__((vector_size(32)));
constexpr unsigned nvdouble = sizeof(vdouble)/sizeof(double);

void fct(vdouble *r,
         const vdouble *a,
         const vdouble *b,
         double f)
{
  for (unsigned i = 0; i < 128 / nvdouble; ++i)
    r[i] = a[i] * f + b[i];
}
```

Direct Accelerator Programming

```
__global__ void fct(double *r,
                    const double *a,
                    const double *b,
                    double f)
{
  int i = blockIdx.x*blockDim.x+threadIdx.x;
  r[i] = a[i] * f + b[i];
}
```

Indirect Accelarator Programming

```
void fct(double *r,
         const double *a,
         const double *b,
         double f) {
#pragma omp offload target(mic)
#pragma omp parallel for
  for (unsigned i = 0; i < 128; ++i)
    r[i] = a[i] * f + b[i];
}
```

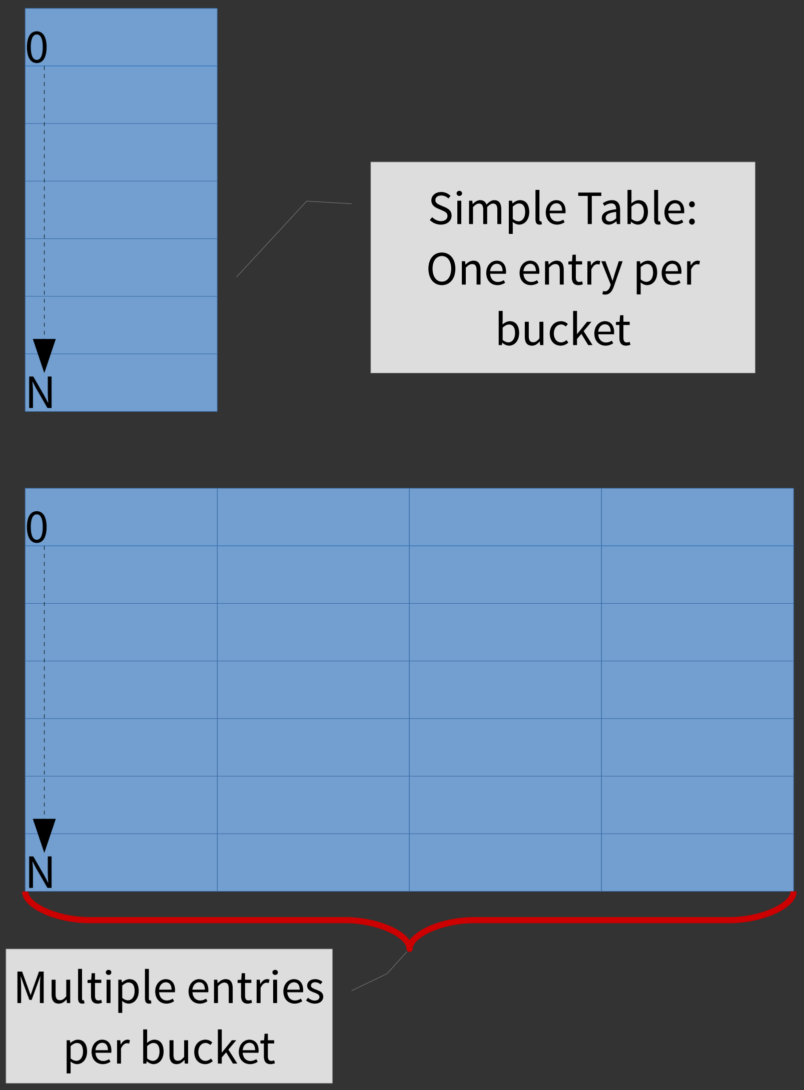
Not Just Arithmetic

- Linear search:

```
bool avail_p(const uint32_t *a, uint32_t v)
{
  for (unsigned i = 0; i < 128; i += 4)
    if (vbitmask(vcmp(a[i], broadcast(v),
                      CMP_EQ)))
      return true;
  return false;
}
```

Not Just Arithmetic

- Perhaps even hash tables
  - Increased locality
- Problem:
  - atomicity



New Algorithm

- Sometimes Algorithm not suited
  - e.g.: Mersenne Twister
- Alternative Algorithm
  - SIMD-oriented Fast Mersenne Twister (SFMT)
  - Mersenne Twister for Graphics Processor (MTGP)

Summary

- Vectorization: not just for vector math
- Many different instructions to solve interesting problems
- Only getting more powerful and important
- Compilers getting good
  - Automatic optimization
  - Direct control
  - Vector types including operations
  - Multi-versioning
- Specialized algorithms

Operations:

- Gather/Scatter
- Conditional operations
  - Mask vectors
  - Masked operations
- Shuffle/Merge operations
- Broadcast, Expand, Compress
- Reduction operations

Questions?