# CSC 252: Computer Organization
# Spring 2018: Lecture 14

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

**Action Items:**
- **Mid-term: March 8 (Thursday)**

# Announcements

- Mid-term exam: March 8; in class.

# Announcements

- Mid-term exam: March 8; in class.

- Prof. Scott has some past exams posted: https://www.cs.rochester.edu/courses/252/spring2014/resources.shtml.
- Mine will be less writing, less explanation.

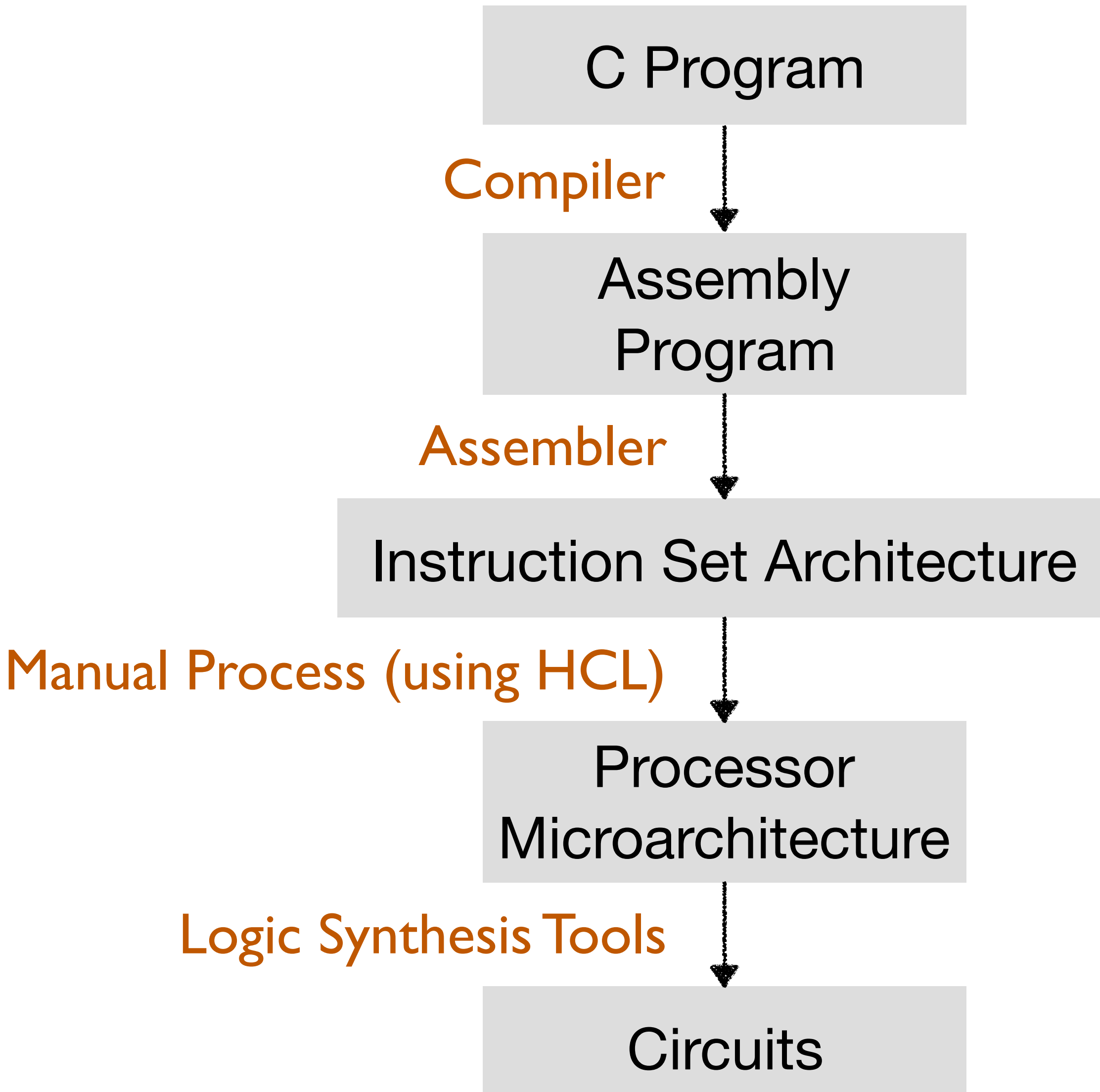
# Announcements

- Mid-term exam: March 8; in class.

- Prof. Scott has some past exams posted: https://www.cs.rochester.edu/courses/252/spring2014/resources.shtml.
- Mine will be less writing, less explanation.

- Open book test: any sort of paper-based product, e.g., book, **notes**, magazine, old tests. I don't think they will help, but it's up to you.
- Exams are designed to test your ability to apply what you have learned and not your memory (though a good memory could help).
- **Nothing electronic**, including laptop, cell phone, calculator, etc.
- **Nothing biological**, including your roommate, husband, wife, your hamster, another professor, etc.
- "**I don't know**" gets15% partial credit. Must cross/erase everything else.

# So far in 252…


C Program
Compiler
Assembly Program
Assembler
Instruction Set Architecture
Manual Process (using HCL)
Processor Microarchitecture
Logic Synthesis Tools
Circuits

# So far in 252…

C Program

↓ Compiler

Assembly Program

↓ Assembler

Instruction Set Architecture

↓ Manual Process (using HCL)

Processor Microarchitecture

↓ Logic Synthesis Tools

Circuits

# Today: Optimizing Code Transformation

- Overview
- Hardware/Microarchitecture Independent Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
- Optimization Blockers
  - Procedure calls
  - Memory aliasing
- Exploit Hardware Microarchitecture

# Optimizing Compilers

- Algorithm choice decides overall complexity (big O), system decides constant factor in the big O notation
- System optimizations don't (usually) improve
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors, but constant factors also matter
- Compilers provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies
- Have difficulty overcoming "optimization blockers"
  - potential memory aliasing
  - potential procedure side-effects

# Today: Optimizing Code Transformation

- Overview
- Hardware/Microarchitecture Independent Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
- Optimization Blockers
  - Procedure calls
  - Memory aliasing
- Exploit Hardware Microarchitecture

# Generally Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor / compiler

- Code Motion
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
         a[n*i+j] = b[j];
}
```

→

```
    long j;
    int ni = n*i;
    for (j = 0; j < n; j++)
         a[ni+j] = b[j];
```

# Compiler-Generated Code Motion (-O1)

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
         a[n*i+j] = b[j];
}
```

```
    long j;
    long ni = n*i;
    double *rowp = a+ni;
    for (j = 0; j < n; j++)
         *rowp++ = b[j];
```

```
set_row:
        testq    %rcx, %rcx                # Test n
        jle      .L1                       # If 0, goto done
        imulq    %rcx, %rdx                # ni = n*i
        leaq     (%rdi,%rdx,8), %rdx       # rowp = A + ni*8
        movl     $0, %eax                  # j = 0
.L3:                                       # loop:
        movsd    (%rsi,%rax,8), %xmm0      # t = b[j]
        movsd    %xmm0, (%rdx,%rax,8)      # M[A+ni*8 + j*8] = t
        addq     $1, %rax                  # j++
        cmpq     %rcx, %rax                # j:n
        jne      .L3                       # if !=, goto loop
.L1:                                       # done:
        rep ; ret
```

# Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  - `16*x` --> `x << 4`
  - Depends on cost of multiply or divide instruction
  - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

# Common Subexpression Elimination

- Reuse portions of expressions
- GCC will do this with –O1

3 multiplications: i*n, (i–1)*n, (i+1)*n

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j  ];
down =  val[(i+1)*n + j  ];
left =  val[i*n     + j-1];
right = val[i*n     + j+1];
sum = up + down + left + right;
```

```
leaq   1(%rsi), %rax   # i+1
leaq   -1(%rsi), %r8   # i-1
imulq  %rcx, %rsi      # i*n
imulq  %rcx, %rax      # (i+1)*n
imulq  %rcx, %r8       # (i-1)*n
addq   %rdx, %rsi      # i*n+j
addq   %rdx, %rax      # (i+1)*n+j
addq   %rdx, %r8       # (i-1)*n+j
```

1 multiplication: i*n

```
long inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

```
imulq    %rcx, %rsi   # i*n
addq     %rdx, %rsi   # i*n+j
movq     %rsi, %rax   # i*n+j
subq     %rcx, %rax   # i*n+j-n
leaq     (%rsi,%rcx), %rcx # i*n+j+n
```

# Today: Optimizing Code Transformation

- Overview
- Hardware/Microarchitecture Independent Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
- Optimization Blockers
  - Procedure calls
  - Memory aliasing
- Exploit Hardware Microarchitecture

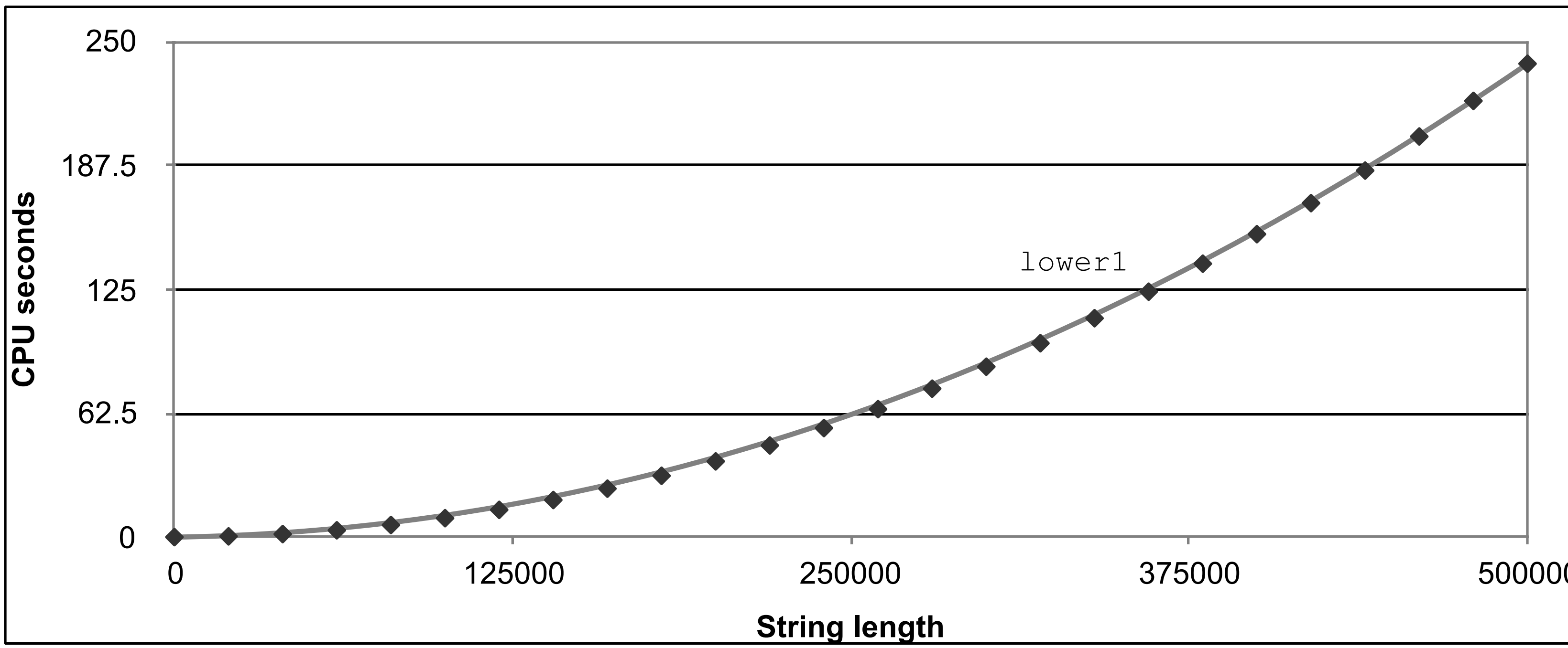
# Optimization Blocker #1: Procedure Calls

- Procedure to Convert String to Lower Case

```
void lower(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

# Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance



# Calling Strlen

```
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
       s++;
       length++;
    }
    return length;
}
```

- Strlen performance
  - Has to scan the entire length of a string, looking for null character.
  - O(N) complexity
- Overall performance
  - N calls to strlen
  - Overall $O(N^2)$ performance

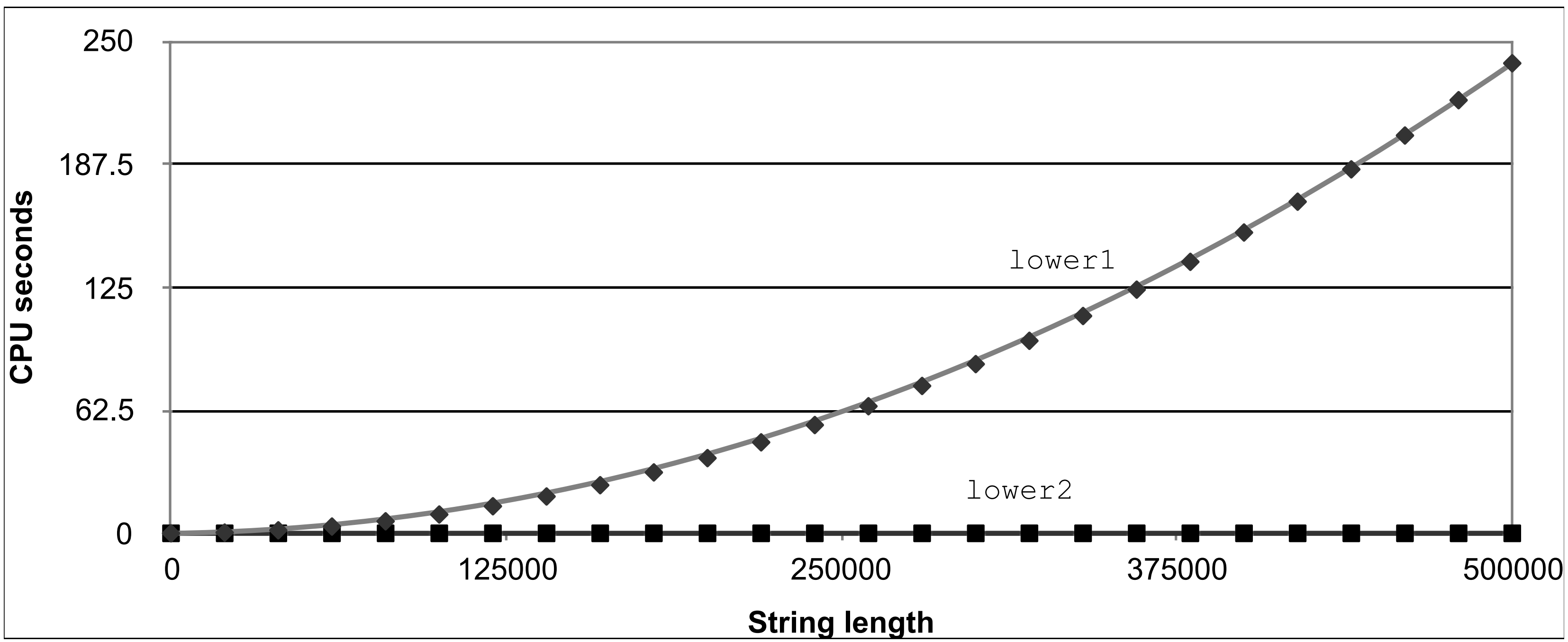
# Improving Performance

- Move call to `strlen` outside of loop
- Since result does not change from one iteration to another
- Form of code motion

```
void lower(char *s)
{
  size_t i;
  size_t len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

# Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance now



# Optimization Blocker: Procedure Calls

```
void lower(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}

size_t total_lencount = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
       s++; length++;
    }
    total_lencount += length;
    return length;
}
```

Why couldn't compiler move `strlen` out of loop?

- Procedure may have side effects, e.g., alters global state each time called
- Function may not return same value for given arguments

# Optimization Blocker: Procedure Calls

- Most compilers treat procedure call as a black box
  - Assume the worst case, Weak optimizations near them
  - There are interprocedural optimizations (IPO), but they are expensive
  - Sometimes the compiler doesn't have access to source code of other functions because they are object files in a library. Link-time optimizations (LTO) comes into play, but are expensive as well.
- Remedies:
  - Use of inline functions
    - GCC does this with –O1, but only within single file
  - Do your own code motion

# Optimization Blocker #2: Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

**Value of `A`:**

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16,
   32,  64, 128};
```

**Value of `B`:**

```
init:  [x, x, x]
```

# Optimization Blocker #2: Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

**Value of `A`:**

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16,
   32,  64, 128};
```

**Value of `B`:**

```
init:   [x, x, x]
```

```
i = 0:  [3, x, x]
```

# Optimization Blocker #2: Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

## Value of `A`:

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16,
   32,  64, 128};
```

## Value of `B`:

```
init:   [x, x, x]
```

```
i = 0:  [3, x, x]
```

```
i = 1:  [3, 28, x]
```

# Optimization Blocker #2: Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

## Value of A:

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16,
   32,  64, 128};
```

## Value of B:

```
init:   [x, x, x]
```

```
i = 0: [3, x, x]
```

```
i = 1: [3, 28, x]
```

```
i = 2: [3, 28, 224]
```

# Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Every iteration updates memory location b[i]

# Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Every iteration updates memory location b[i]

```
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
```

Every iteration updates `val`, which could stay in register

# Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Every iteration updates memory location b[i]

```
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
```

Every iteration updates `val`, which could stay in register

Why can't a compiler perform this optimization?

# Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

**Value of A:**

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16,
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

B

**Value of B:**

```
init:  [4, 8, 16]
```

# Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

**Value of `A`:**

B →

```
double A[9] =
  { 0,   1,   2,
    4,   8,   16,
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

**Value of `B`:**

```
init:   [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

# Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

**Value of A:**

B

```
double A[9] =
  { 0,   1,   2,
    3,   8,  16,
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

**Value of B:**

```
init:   [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

# Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

## Value of A:

B

```
double A[9] =
  { 0,   1,   2,
    3,   0,  16,
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

## Value of B:

```
init:  [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

# Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

**Value of A:**

```
double A[9] =
  { 0,   1,   2,
    3,   3,  16,
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

B

**Value of B:**

```
init:  [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

# Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

**Value of A:**

B

```
double A[9] =
  { 0,   1,   2,
    3,   6,  16,
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

**Value of B:**

```
init:   [4, 8, 16]
```

```
i = 0:  [3, 8, 16]
```

# Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

## Value of A:

B

```
double A[9] =
  { 0,   1,   2,
    3,  22,  16,
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

## Value of B:

```
init:   [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

# Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

**Value of `A`:**

B

```
double A[9] =
  { 0,   1,    2,
    3,  22,   16,
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

**Value of `B`:**

```
init:   [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

```
i = 1: [3, 22, 16]
```

# Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

**Value of A:**

B

```
double A[9] =
  { 0,   1,   2,
    3,  22,  16,
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

**Value of B:**

```
init:   [4, 8, 16]

i = 0: [3, 8, 16]

i = 1: [3, 22, 16]

i = 2: [3, 22, 224]
```

# Optimization Blocker: Memory Aliasing

- Aliasing
  - Two different memory references specify single location
  - Easy to have happen in C
    - Since allowed to do address arithmetic
    - Direct access to storage structures
  - Get in habit of introducing local variables
    - Accumulating within loops
    - Your way of telling compiler not to check for aliasing

# Today: Optimizing Code Transformation

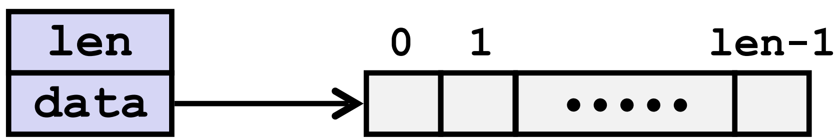
- Overview
- Hardware/Microarchitecture Independent Optimizations
    - Code motion/precomputation
    - Strength reduction
    - Sharing of common subexpressions
- Optimization Blockers
    - Procedure calls
    - Memory aliasing
- Exploit Hardware Microarchitecture

# Exploiting Instruction-Level Parallelism

- Hardware can execute multiple instructions in parallel
  - Pipeline is a classic technique
- Performance limited by data dependencies
- Simple transformations can yield dramatic performance improvement
  - Compilers often cannot make these transformations
  - Lack of associativity and distributivity in floating-point arithmetic

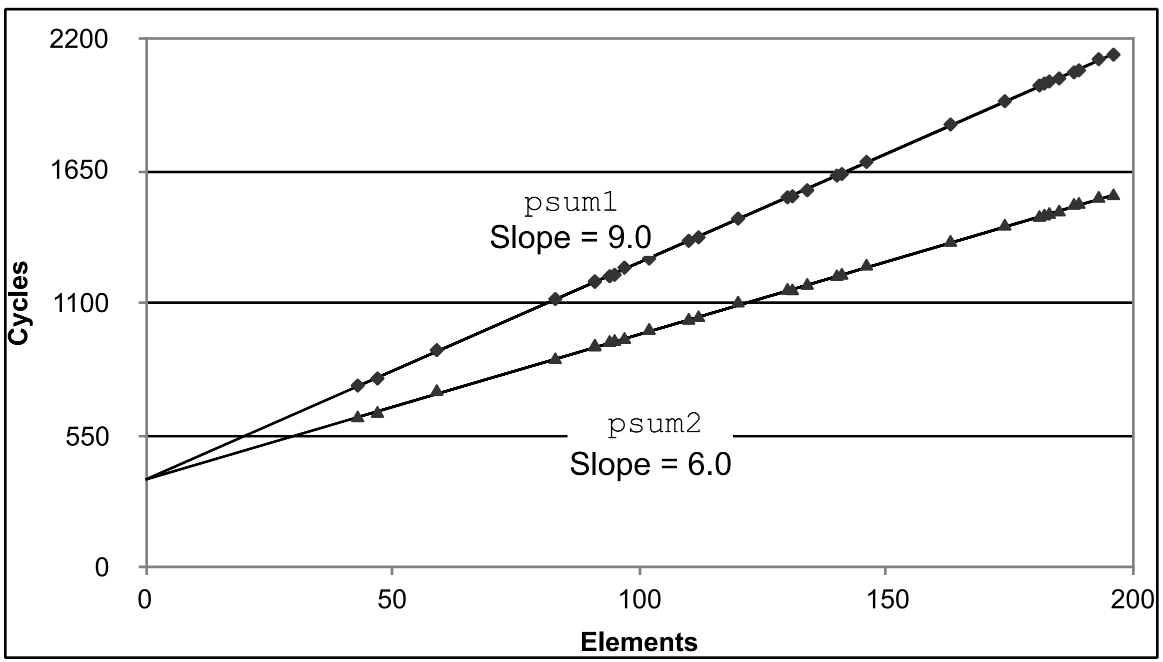
# Running Example: Combine Vector Elements

```
/* data structure for vectors */
typedef struct{
    size_t len;
    int *data;
} vec;
```



# Cycles Per Element (CPE)

- Convenient way to express performance of program that operates on vectors or lists
- T = CPE*n + Overhead, where n is the number of elements
  - CPE is slope of line



# Vanilla Version

```
void combine1(vec_ptr v, int *dest)
{
     long int i;
     *dest = IDENT;
     for (i = 0; i < vec_length(v); i++) {
        int val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
     }
}
```

```
/* retrieve vector element
   and store at val */
int get_vec_element
  (*vec v, size_t idx, int *val)
{
    if (idx >= v->len)
         return 0;
    *val = v->data[idx];
    return 1;
}
```

| OP | Add | Mult |
|---|---|---|
| Combine1 –O1 | 10.12 | 10.12 |

# Basic Optimizations

```
void combine4(vec_ptr v, data_t
*dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

- Move `vec_length` out of loop
- Avoid bounds check on each iteration in `get_vec_element`
- Accumulate in temporary

# Basic Optimizations

```
void combine4(vec_ptr v, data_t
*dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

- Move `vec_length` out of loop
- Avoid bounds check on each iteration in `get_vec_element`
- Accumulate in temporary

| Operation | Add | Mult |
|---|---|---|
| Combine1 –O1 | 10.12 | 10.12 |
| Combine4 | 1.27 | 3.01 |
| Operation Latency | 1.00 | 3.00 |

# x86-64 Compilation of Combine4 Inner Loop

```
for (i = 0; i < length; i++) {
  t = t * d[i];
  *dest = t;
}
```

```
.L519:
  imulq (%rax,%rdx,4), %ecx
  addq  $1, %rdx          # i++
  cmpq  %rdx, %rbp        # Compare length:i
  jg    .L519             # If >, goto Loop
```

Real work

Overhead

| Operation | Add | Mult |
|---|---|---|
| Combine4 | 1.27 | 3.01 |
| Operation Latency | 1.00 | 3.00 |

# Loop Unrolling (2x1)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Perform 2x more useful work per iteration
- Reduce loop overhead (comp, jmp, index dec, etc.)

# Effect of Loop Unrolling

- Helps integer add
  - Approaches latency limit
- But not integer multiply
  - Why?
  - Mult had already approached the limit

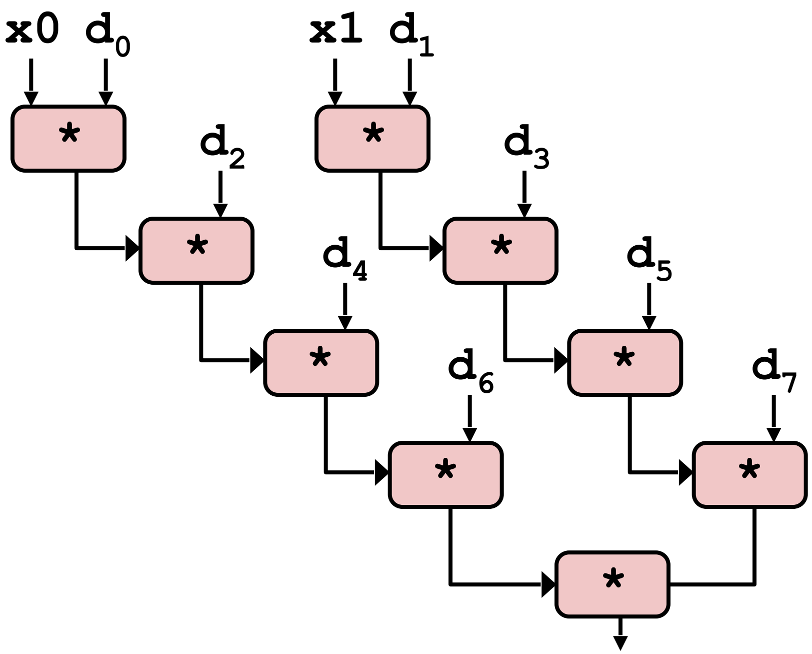
| Operation | Add | Mult |
|---|---|---|
| Combine4 | 1.27 | 3.01 |
| Unroll 2x1 | 1.01 | 3.01 |
| Operation Latency | 1.00 | 3.00 |

# Loop Unrolling with Separate Accumulators

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x0 = x0 OP d[i];
       x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

# Separate Accumulators

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

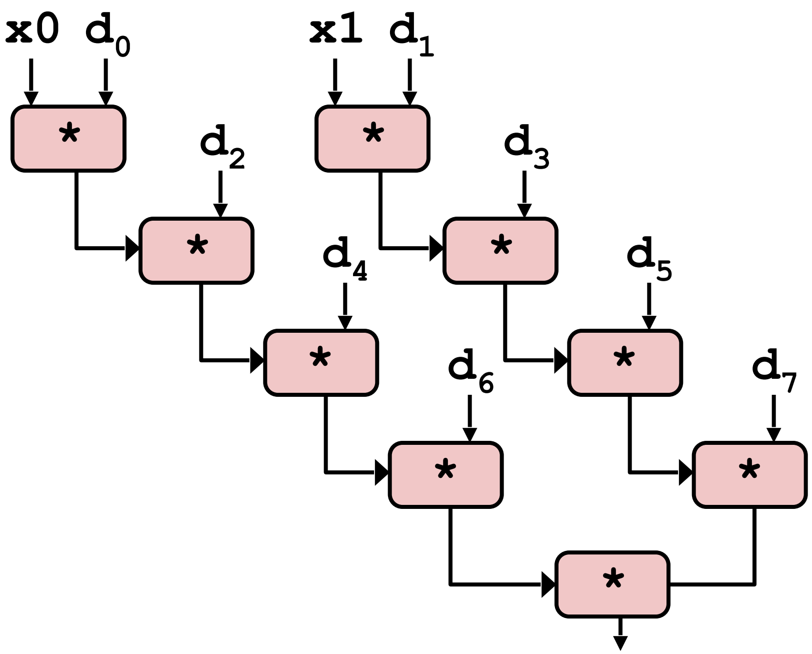



- What changed:
  - Two independent “streams” of operations

- Overall Performance
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles: CPE = D/2

# Separate Accumulators

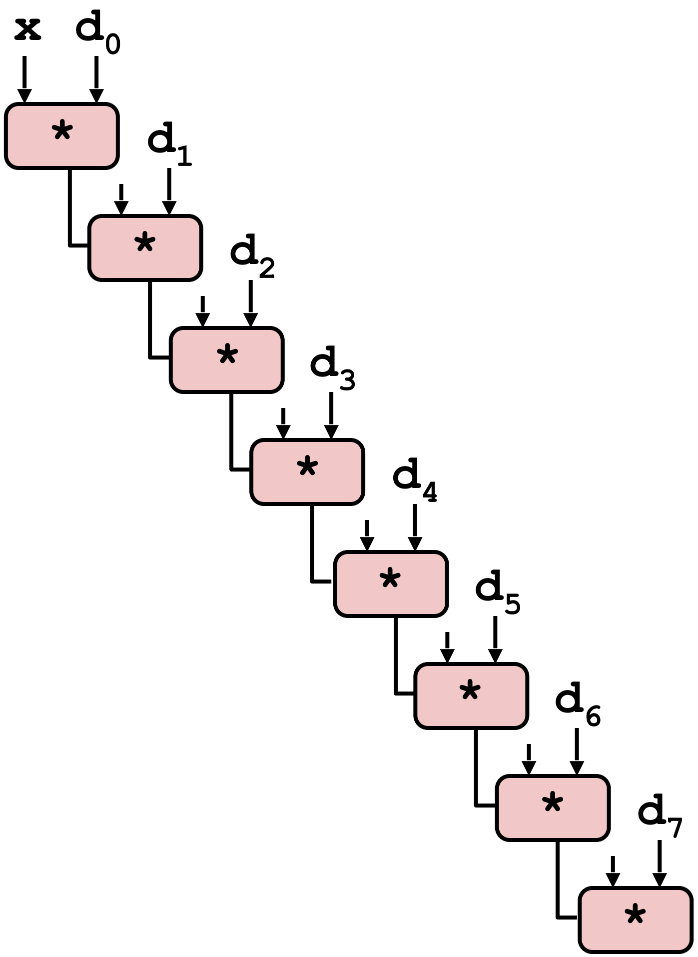
```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```



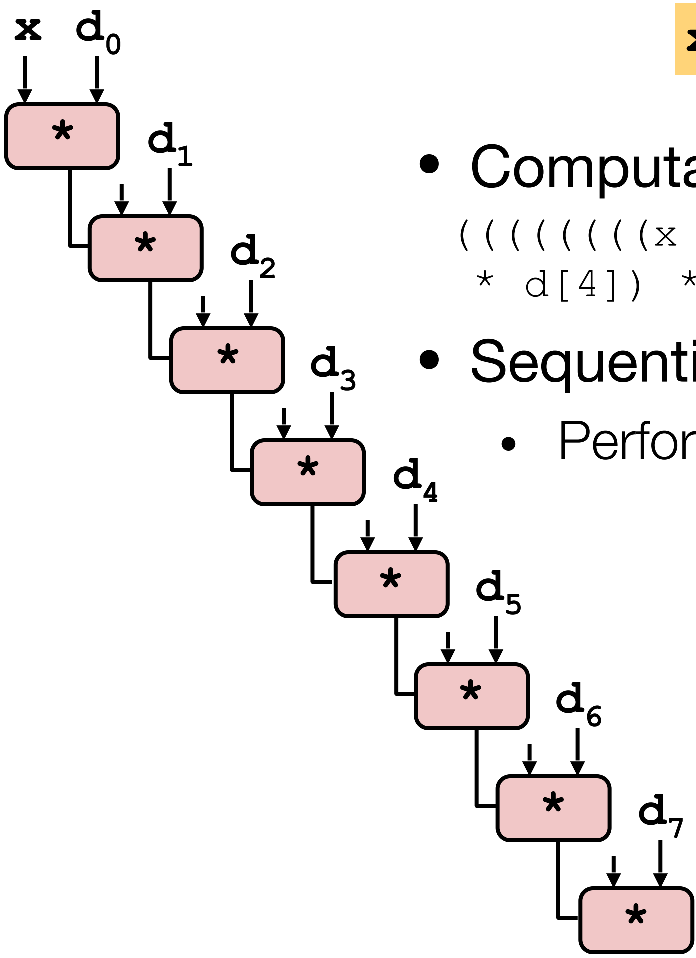

- What changed:
  - Two independent “streams” of operations
- Overall Performance
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles: CPE = D/2

| Operation | Add | Mult |
|---|---|---|
| Combine4 | 1.27 | 3.01 |
| Unroll 2x1 | 1.01 | 3.01 |
| Unroll 2x2 | 0.81 | 1.51 |
| Operation Latency | 1.00 | 3.00 |

# Combine4 = Serial Computation (OP = *)



# Combine4 = Serial Computation (OP = *)




```
x = (x OP d[i]) OP d[i+1];
```

- Computation (length=8)

```
((((((((x * d[0]) * d[1]) * d[2]) * d[3])
 * d[4]) * d[5]) * d[6]) * d[7])
```

- Sequential dependence
  - Performance: determined by latency of OP

# Loop Unrolling with Reassociation

```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

Before

```
x = (x OP d[i]) OP d[i+1];
```

# Loop Unrolling with Reassociation

```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

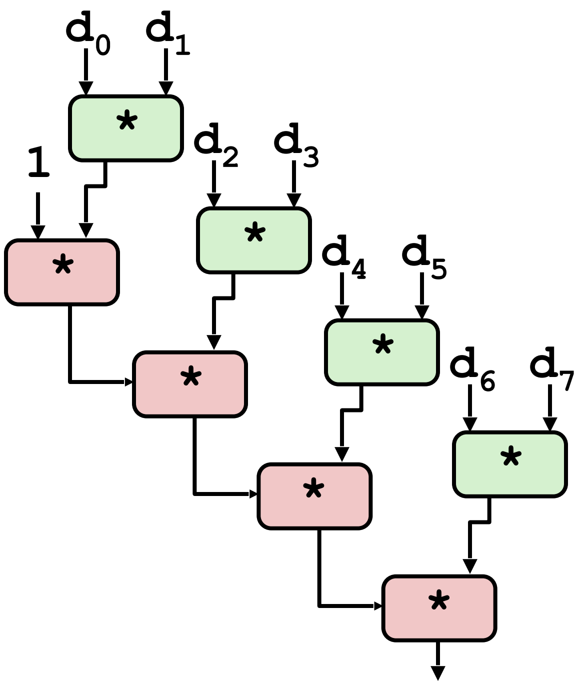
Before

```
x = (x OP d[i]) OP d[i+1];
```

Not always accurate for floating point.

# Reassociated Computation

```
x = x OP (d[i] OP d[i+1]);
```

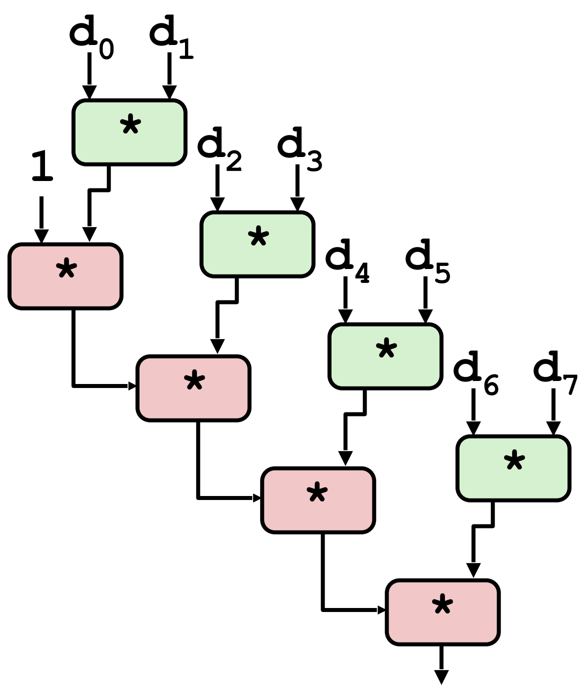



- What changed:
  - Ops in the next iteration can be started early (no dependency)

- Overall Performance
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles: CPE = D/2

# Reassociated Computation

```
x = x OP (d[i] OP d[i+1]);
```




- What changed:
  - Ops in the next iteration can be started early (no dependency)
- Overall Performance
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles:
    CPE = D/2

| Operation | Add | Mult |
|---|---|---|
| Combine4 | 1.27 | 3.01 |
| Unroll 2x1 | 1.01 | 3.01 |
| Unroll 2x1a | 1.01 | 1.51 |
| Operation Latency | 1.00 | 3.00 |