CS356 Unit 13

Performance

Compiling with Optimizations

- Compilers usually have options to apply optimization
- Example: `gcc/g++ -O`*n*
  - `-O0`: No optimization (the default); generates unoptimized code but has the fastest compilation time.
  - `-O1`: Moderate optimization; optimizes reasonably well but does not degrade compilation time significantly.
  - `-O2`: Full optimization; generates highly optimized code and has the slowest compilation time.
  - `-O3`: Full optimization as in -O2; also uses more aggressive automatic inlining of subprograms within a unit and attempts to vectorize loops.
  - `-Os`: Optimize space usage (code and data) of resulting program.

- However, there are still many things the programmer can do to help

https://gcc.gnu.org/onlinedocs/gnat_ugn/Optimization-Levels.html

Profiling

- **Rule: Optimize the common case**
  - A small optimization in code that accounts for a large amount of execution time is worth far more than a large optimization in code that accounts for a small fraction of the execution time
- Q: How do you know where time is being spent?
- A: Profilers!
  - Instrument your code to take statistics as it runs and then can show you what percentage of time each function or even line of code was responsible for
  - Common profilers
    - `gprof` (usually standard with Unix / Linux installs) and `gcc/g++`
    - Intel VTune
    - MS Visual Studio Profiling Tools

```
void someTask( /* args */ )
{
  /* Segment A – sequential code */

  for(int i=0; i<N; i++){
    /* Segment B */
    for(int j=0; j<N; j++){
      /* Segment C */
    }
  }
  return 0;
}
```

Which code segment should you likely focus your time optimizing?

gprof Output

- To instrument your code for profiling:
  - `$ gcc -pg prog1.c -o prog1`
- Run your code
  - `./prog1`
  - This will run the program and generate a file with statistics: `gmon.out`
- Process the profiler results
  - `$ gprof prog1 gmon.out > results.txt`
  - View `results.txt`

```
%   cumulative   self              self     total
time   seconds   seconds    calls   s/call   s/call  name
42.96      4.48     4.48 56091649     0.00     0.00  Board::operator<(Board const&) const
 6.43      5.15     0.67  2209524     0.00     0.00  std::_Rb_tree<...>::_M_lower_bound(...)
 5.08      5.68     0.53 108211500    0.00     0.00  __gnu_cxx::__normal_iterator<...>::operator+(...)
 4.51      6.15     0.47  4419052     0.00     0.00  Board::Board(Board const&)
 4.32      6.60     0.45  1500793     0.00     0.00  void std::__adjust_heap<...>(...)
 3.84      7.00     0.40 28553646     0.00     0.00  PuzzleMove::operator>(PuzzleMove const&) const
```

OPTIMIZATION BLOCKERS

Reducing Function Calls

- Consider the "original" code to the right
- Can we optimize by converting the original code to the proposed optimized code?
  - No!
- Functions may have side effects
  - What if x is incremented in the function

```
#include <stdio.h>

int x=0;

int f1()
{
  /* Produces & returns an int */
  return ++x;
}

int main()
{
  int y = f1() + f1() + f1() + f1();
  printf("%d\n", y);
  return 0;
}
```

Original Code

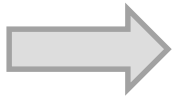
```
#include <iostream>
using namespace std;

...

int main()
{
   int y = 4*f();
   cout << y << endl;
   return 0;
}
```

Proposed Optimization

Function Inlining

- Inlining is the process of substituting the function code into each location where it is called
- This avoids the overhead of a function call at the cost of greater code size (duplication)
  - **Note: Compiling with optimization levels above -O0 allow the compiler to auto-inline functions of its choice (usually small functions)**

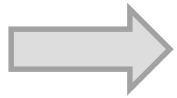
```
int x=0;

int f1()
{
  /* Produces & returns an int */
  return ++x;
}

int main()
{
  int y = f1() + f1() + f1() + f1();
  printf("%d\n", y);
  return 0;
}
```



```
int x=0;

int f1()
{
  /* Produces & returns an int */
  return ++x;
}

int main()
{
  int y = ++x + ++x + ++x + ++x;
  printf("%d\n", y);
  return 0;
}
```

Inlining

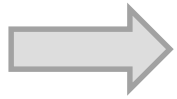
```
int x=0;

int f1()
{
  /* Produces & returns an int */
  return ++x;
}

int main()
{
  int y = f1() + f1() + f1() + f1();
  printf("%d\n", y);
  return 0;
}
```



g++ -O1 …

```
main:
  ...
  movl    $0, %eax
  call    f1
  movl    %eax, %ebx
  movl    $0, %eax
  call    f1
  addl    %eax, %ebx
  movl    $0, %eax
  call    f1
  addl    %eax, %ebx
  movl    $0, %eax
  call    f1
  leal    (%rbx,%rax), %edx
```

```
int x=0;

int f1()
{
  /* Produces & returns an int */
  return ++x;
}

int main()
{
  int y = f1() + f1() + f1() + f1();
  printf("%d\n", y);
  return 0;
}
```

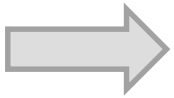


g++ -O2 …

```
main:
  ...
  movl    x(%rip), %edx    # %edx = x
  leal    4(%rdx), %eax    # %eax = 4+x
  movl    %eax, x(%rip)    # x = 4+x
  leal    6(%rdx,%rdx,2), %edx  # %edx=3x+6
  addl    %eax, %edx            # %edx=4x+10
```

Inlining

```
int f1(vector<int>& v1)
{
  int total = 0;
  for(int i=0; i < v1.size(); i++){
    total += v1[i];
  }
  return total;
}
```



g++ -O1 …

```
_Z2f1RSt6vectorIiSaIiEE:
.LFB509:
        .cfi_startproc
        movq    (%rdi), %rsi
        movq    8(%rdi), %rax
        subq    %rsi, %rax
        sarq    $2, %rax
        movq    %rax, %rdi
        testq   %rax, %rax
        je      .L4
        movl    $0, %ecx
        movl    $0, %edx
        movl    $0, %eax
.L3:
        addl    (%rsi,%rcx,4), %eax
        addl    $1, %edx
        movslq  %edx, %rcx
        cmpq    %rdi, %rcx
        jb      .L3
        rep ret
.L4:
        movl    $0, %eax
        ret
```

Notice there is no call to vector's size function. Compiling with optimization levels –O0 would cause it to NOT inline the call

Limits of Inlining

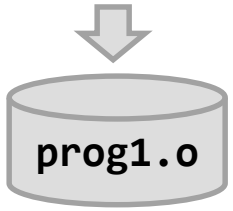
- Inlining can only be done when the definition of the function is in the same translation unit
  - Recall the compiler only sees the code in the current translation unit (file) and so won't see the definition of f1() in lib.c to be able to inline it

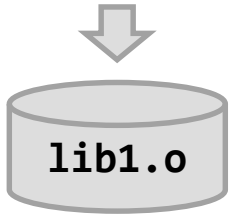
prog1.c

```
extern int x;

int f1();

int main()
{
  int y = f1() + f1() + f1() + f1();
  printf("%d\n", y);
  return 0;
}
```

lib1.c

```
int x=0;

int f1()
{
  /* Produces & returns an int */
  return ++x;
}
```





C++ Templates and Inlining

- Since .h files are #include'd, any functions defined in the .h file can then be inlined
- This is one reason templates offer some advantage in C++ is because their definition is ALWAYS available

prog1.c

```
#include "vec.h"

int main()
{
  vec<int> myvec;
  for(int i=0; i < myvec.size(); i++){
    ...
  }
  ...
}
```

vec.h

```
template<typename T>
class vec
{
public:
   ...
   int size() const;
private:
   int size_;
};

template <typename T>
int vec<T>::size() const
{ return size_; }
```

Memory Aliasing

- Consider twiddle1 and its function to return x + 2y
- Now suppose we have pointers as arguments
  - We could write twiddle2a (to try to do what twiddle1 did)
  - Is it equivalent to twiddle1?
  - Is twiddle2b equivalent to twiddle2a?
- No!
  - Not if xp and yp point to the same value.

```
int twiddle1(long x, long y)
{
  x += y;
  x += y;
  return x;  // x + 2*y
}

// Now with pointers
int twiddle2a(long* xp, long* yp)
{
  *xp += *yp;
  *xp += *yp;
  return *xp;
}

int twiddle2b(long* xp, long* yp)
{
  *xp += 2 * (*yp);
  return *xp;
}

int ans = 0;
void f1(long x, long y)
{
  ans = twiddle1(x,y);
  ans += twiddle2a(&x,&y);
  ans += twiddle2b(&x,&y);
}
```

Memory Aliasing

- The compiler must play it safe and generate code that would work if both pointers contain the same address (i.e. reference the same variable)…we call this memory aliasing

```
// Notice the compiler optimized
// to perform x + 2*y
twiddle1:
        leaq    (%rdi,%rsi,2), %rax
        ret

// But here it left it as two
// separate adds
twiddle2a:
        movq    (%rsi), %rax
        addq    (%rdi), %rax
        movq    %rax, (%rdi)
        addq    (%rsi), %rax
        movq    %rax, (%rdi)
        ret
```

```
int twiddle1(long x, long y)
{
  x += y;
  x += y;
  return x;  // x + 2*y
}

// Now with pointers
int twiddle2a(long* xp, long* yp)
{
  *xp += *yp;
  *xp += *yp;
  return *xp;
}

int twiddle2b(long* xp, long* yp)
{
  *xp += 2 * (*yp);
  return *xp;
}

int ans = 0;
void f1(long x, long y)
{
  ans = twiddle1(x,y);
  ans += twiddle2a(&x,&y);
  ans += twiddle2b(&x,&y);
}
```

Memory Aliasing

- Aliasing may also affect inlining
  - -O1 does not inline twiddle2a
  - Running -O3 does end up inlining twiddle2a

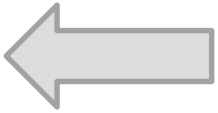
Inlined

Not Inlined

Inlined

```
f1:
  subq    $16, %rsp
  movq    %rdi, 8(%rsp)
  movq    %rsi, (%rsp)

  leaq    (%rdi,%rsi,2), %rax
  movl    %eax, ans(%rip)

  movq    %rsp, %rsi
  leaq    8(%rsp), %rdi
  call    twiddle2a
  movq    8(%rsp), %rdx

  movq    (%rsp), %rcx
  leaq    (%rdx,%rcx,2), %rdx
  addl    ans(%rip), %eax
  addl    %edx, %eax
  movl    %eax, ans(%rip)
  addq    $16, %rsp
  ret
```

gcc -O1 …



```
int twiddle1(long x, long y)
{
  x += y;
  x += y;
  return x;  // x + 2*y
}

// Now with pointers
int twiddle2a(long* xp, long* yp)
{
  *xp += *yp;
  *xp += *yp;
  return *xp;
}

int twiddle2b(long* xp, long* yp)
{
  *xp += 2 * (*yp);
  return *xp;
}

int ans = 0;
void f1(long x, long y)
{
  ans = twiddle1(x,y);
  ans += twiddle2a(&x,&y);
  ans += twiddle2b(&x,&y);
}
```

MAXIMIZING PERFORMANCE

Overview

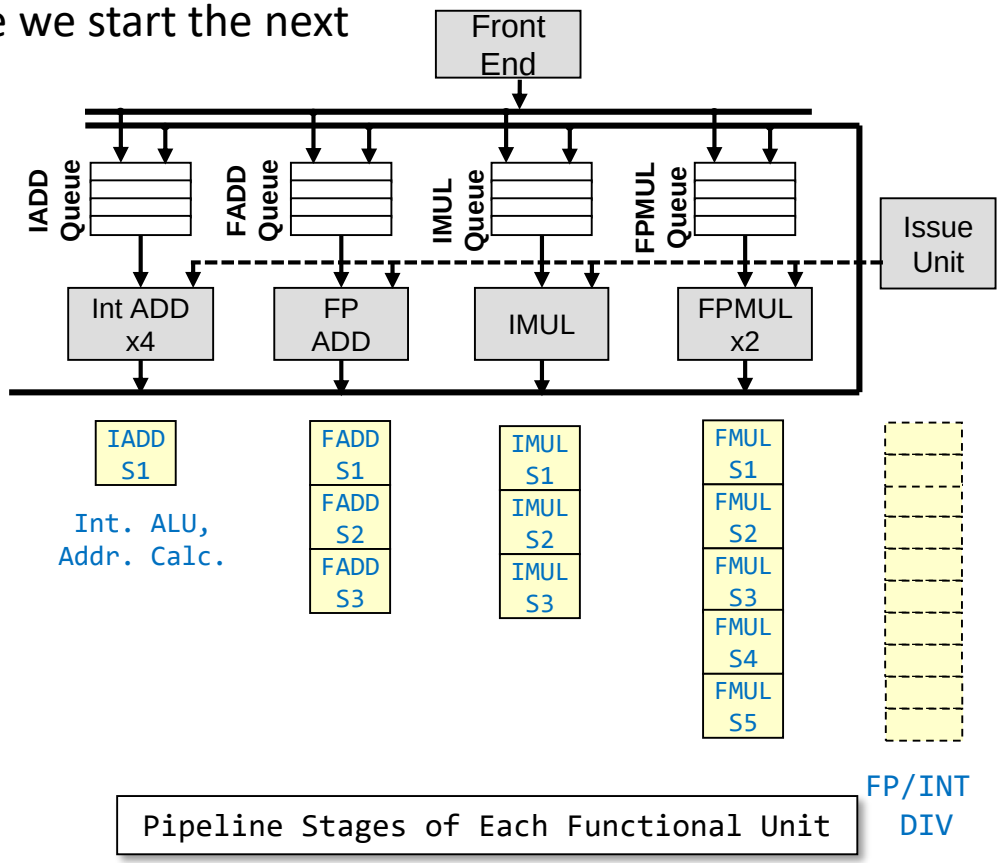
- We have seen our processors have great capability to perform many operations in parallel
- How can we write our code in such a way as to take advantage of those capabilities?
- Are there limits on how much performance we can achieve and how would we know if we are hitting those limits?
- Let's first understand our hardware capabilities

Latency and Throughput (Issue Time)

- Latency: clock cycles (pipeline stages) in the pipeline of that unit
- Some units are not pipelined (ex. Int and FP Divider) which means we cannot overlap operations
  - Must wait for one to finish before we start the next

Functional Unit	Latency (Required stalls cycles between dependent [RAW] instrucs.)	Issue Time (Reciprocal throughput) (Cycles between 2 independent instructions requiring the same FU)
Int ADD	1	1
Int MUL	3	1
FP ADD	3	1
FP Mul.	5	1
FP Div.	3-15	3-15

Latency and Issue times for Intel's Haswell architecture



Combine1 (Base)

- Base implementation of combining elements of an array/vector
- Use MACROS to be able to easily switch between adding and multiplying
- Use typedefs to be able to switch types: `int` or `float`
- Attempt to measure clock cyles per element (CPE)
  - Total clock cycles / array size

```
#define OP +
#define IDENT 0
#ifndef FP
typedef int DTYPE;
#else
typedef double DTYPE;
#endif


void combine1(struct vec* v1, DTYPE* tot)
{
  DTYPE* data = v1->dat;
  *tot = IDENT;
  for(unsigned i = 0; i < get_size(v1); i++)
  {
    *tot = *tot OP data[i];
  }
}
```

```
struct vec {
  DTYPE* dat;
  unsigned size;
};
```

combine1.c (Base)

Function	Method	Int (+)	Int (*)	FP (+)	FP (*)
combine1	Base	10.12	10.12	10.17	11.14

Performance (CPE = Clocks Per Element)

Combine2 (Code Motion)

- No need to repeat call to `get_size()` each call
- Code Motion
  - Move code outside the loop

```
#define OP +
#define IDENT 0
#ifndef FP
typedef int DTYPE;
#else
typedef double DTYPE;
#endif

void combine2(struct vec* v1, DTYPE* tot)
{
  DTYPE* data = v1->dat;
  *tot = IDENT;
  unsigned size = get_size(v1);
  for(unsigned i = 0; i < size; i++)
  {
    *tot = *tot OP data[i];
  }
}
```

combine2.c
(Move get_size)

Function	Method	Int (+)	Int (*)	FP (+)	FP (*)
combine1	Base	10.12	10.12	10.17	11.14
Combine2	Move get_size	7.02	9.03	9.02	11.03

Combine4 (Use temporary)

- Use a temporary accumulator variable
- Avoid combine2's memory read and write of `*dest` in each loop iterations
- Why didn't the compiler infer this optimization in combine 1 or 2?
  - Memory aliasing!
  - What if `tot` points to one of the vector elements
  - Ex. [2 3 5] and tot points to 5

```
#define OP +
#define IDENT 0
#ifndef FP
typedef int DTYPE;
#else
typedef double DTYPE;
#endif

void combine4(struct vec* v1, DTYPE* tot)
{
  *tot = IDENT;
  unsigned size = get_size(v1);
  DTYPE* data = v1->dat;
  DTYPE acc = IDENT;
  for(unsigned i = 0; i < size; i++){
    acc = acc OP data[i];
  }
  *tot = acc;
}
```

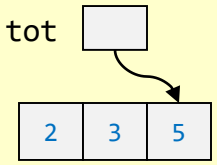
combine4.c
(Temp. accumumulator)

Function	Method	Int (+)	Int (*)	FP (+)	FP (*)
combine2	Move get_size	7.02	9.03	9.02	11.03
combine4	Temp acc.	1.27	3.01	3.01	5.01

Combine4 vs. Combine2

- Why didn't the compiler infer this optimization in combine 1 or 2?
  - Memory aliasing!
  - What if `tot` points to one of the vector elements
  - Ex. [2 3 5] and tot points to 5
  - Combine2: `[2 3 1]  => [2 3 2]  => [2 3 6]  => [2 3 36]`
  - Combine4: `[2 3 5], 1 => [2 3 5], 2 => [2 3 5], 6 => [2 3 5], 30`

```
void combine2(struct vec* v1, DTYPE* tot)
{
  DTYPE* data = v1->dat;
  *tot = IDENT;
  unsigned size = get_size(v1);
  for(unsigned i = 0; i < size; i++)
  {
    *tot = *tot OP data[i];
  }
}
```



combine2.c

```
void combine4(struct vec* v1, DTYPE* tot)
{
  *tot = IDENT;
  unsigned size = get_size(v1);
  DTYPE* data = v1->dat;
  DTYPE acc = IDENT;
  for(unsigned i = 0; i < size; i++){
    acc = acc OP data[i];
  }
  *tot = acc;
}
```

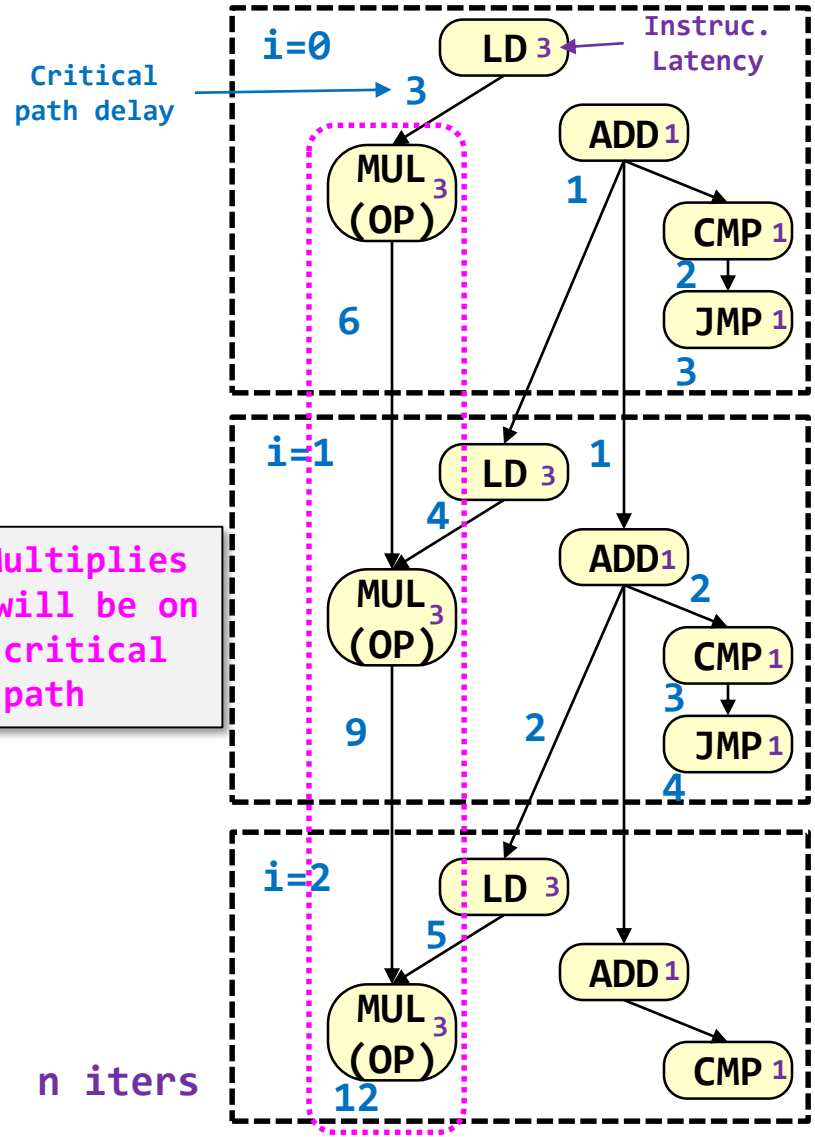
combine4.c

Combine4 Dataflow

```
void combine4(struct vec* v1, DTYPE* tot)
{
  *tot = IDENT;
  unsigned size = get_size(v1);
  DTYPE* data = v1->dat;
  DTYPE acc = IDENT;
  for(unsigned i = 0; i < size; i++){
    acc = acc OP data[i];
  }
  *tot = acc;
}
```

```
.L3:
        mulsd   (%rdx), %xmm0
        addq    $8, %rdx
        cmpq    %rax, %rdx
        jne     .L3

.L3:
        ld      (%rdx), %xmm1
        mulsd   %xmm1, %xmm0
        addq    $8, %rdx
        cmpq    %rax, %rdx
        jne     .L3
```

Add/Multiplies (OP) will be on the critical path



Combine5 (Loop Unrolling)

- Can we use loop unrolling to try to shorten the critical path
- For the code to the right we see little improvement
- Let's look at the dataflow graph

```
void combine5(struct vec* v1, DTYPE* tot)
{
  *tot = IDENT;
  unsigned size = get_size(v1);
  DTYPE* data = v1->dat;
  DTYPE acc = IDENT;
  unsigned limit = size-1;
  unsigned i;
  for(i = 0; i < limit; i+=2){
    acc = (acc OP data[i]) OP data[i+1];
  }
  for( ; i < size; i++){
    acc = acc OP data[i];
  }
  *tot = acc;
}
```

combine5.c
(Unrolled 2x w/ 1 accumulators)

2x1 =
Times Unrolled x # Accumulators

Function	Method	Int (+)	Int (*)	FP (+)	FP (*)
combine4	Temp acc.	1.27	3.01	3.01	5.01
combine5	Unrolled 2x1	1.01	3.01	3.01	5.01
Ideal	Latency:1/Throughput	1 : 0.5	3 : 1	3 : 1	5 : 0.5

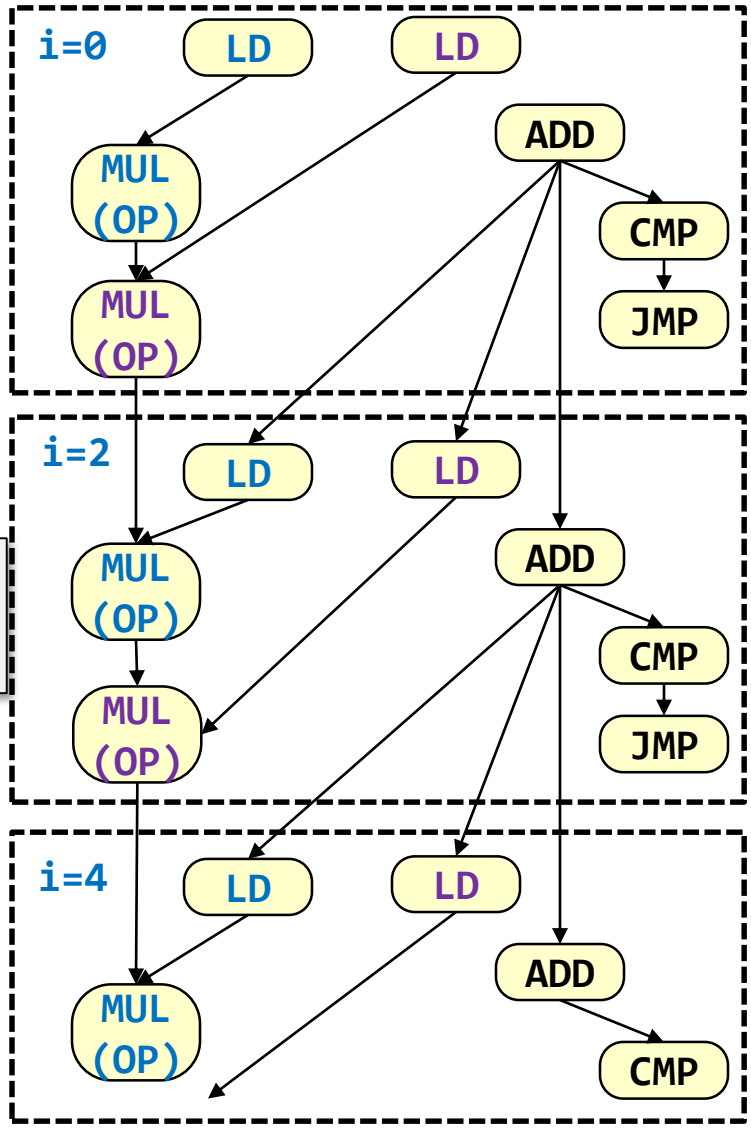
Combine5 Dataflow

```
void combine5(struct vec* v1, DTYPE* tot)
{
  *tot = IDENT;
  unsigned size = get_size(v1);
  DTYPE* data = v1->dat;
  DTYPE acc = IDENT;
  unsigned limit = size-1;
  unsigned i;
  for(i = 0; i < limit; i+=2){
    acc = (acc OP data[i]) OP data[i+1];
  }
  for( ; i < size; i++){
    acc = acc OP data[i];
  }
  *tot = acc;
}
```

```
.L3:
        ld      (%rcx), %xmm1
        mulsd   %xmm1, %xmm0
        ld      8(%rcx), %xmm2
        mulsd   %xmm2, %xmm0
        addl    $16, %rcx
        cmpl    %edx, %edi
        ja      .L3
```

Still n multiplies on critical path

n/2 iters



Combine6 (Loop Unrolling)

- We want to shorten the critical path so we can use 2 separate accumulators

```
void combine6(struct vec* v1, DTYPE* tot)
{
  *tot = IDENT;
  unsigned size = get_size(v1);
  DTYPE* data = v1->dat;
  DTYPE acc0 = IDENT;
  DTYPE acc1 = IDENT;
  unsigned limit = size-1;
  unsigned i;
  for(i = 0; i < limit; i+=2){
    acc0 = acc0 OP data[i];
    acc1 = acc1 OP data[i+1];
  }
  for( ; i < size; i++){
    acc0 = acc0 OP data[i];
  }
  *tot = acc0 OP acc1;
}
```

combine6.c
(Unrolled 2x w/ 2 accumulators)

2x2 =
Times Unrolled x # Accumulators

Function	Method	Int (+)	Int (*)	FP (+)	FP (*)
combine4	Temp acc.	1.27	3.01	3.01	5.01
combine6	Unrolled 2x2	0.81	1.51	1.51	2.51
Ideal	Latency:1/Throughput	1 : 0.5	3 : 1	3 : 1	5 : 0.5

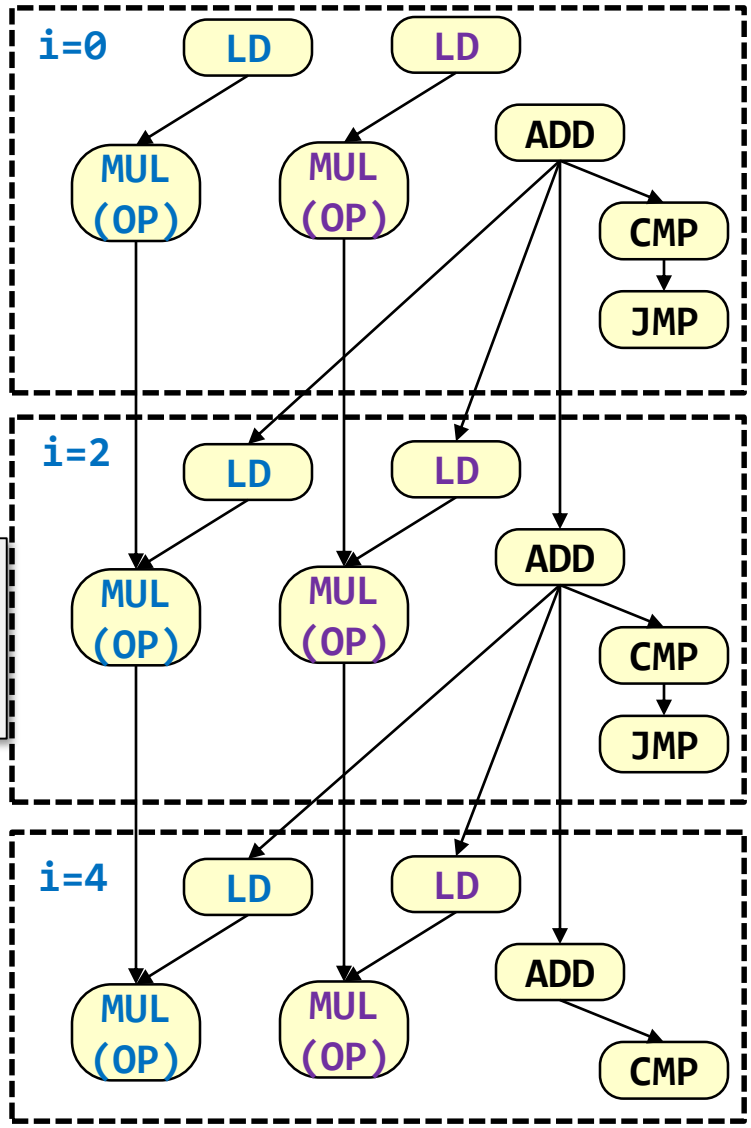
Combine6 Dataflow

```
void combine6(struct vec* v1, DTYPE* tot)
{
  *tot = IDENT;
  unsigned size = get_size(v1);
  DTYPE* data = v1->dat;
  DTYPE acc0 = IDENT;
  DTYPE acc1 = IDENT;
  unsigned limit = size-1;
  unsigned i;
  for(i = 0; i < limit; i+=2){
    acc0 = acc0 OP data[i];
    acc1 = acc1 OP data[i+1];
  }
  for( ; i < size; i++){
    acc0 = acc0 OP data[i];
  }
  *tot = acc0 OP acc1;
}
```

```
.L3:
        ld      (%rcx), %xmm2
        mulsd   %xmm2, %xmm0
        ld      8(%rcx), %xmm3
        mulsd   %xmm3, %xmm1
        addl    $16, %rcx
        cmpl    %edx, %edi
        ja      .L3
```

Critical path shortened to n/2 multiplies.

n/2 iters



Combine5 vs. Combine7

- Why do the following perform so differently?

```
void combine5(struct vec* v1, DTYPE* tot)
{
  *tot = IDENT;
  unsigned size = get_size(v1);
  DTYPE* data = v1->dat;
  DTYPE acc = IDENT;
  unsigned limit = size-1;
  unsigned i;
  for(i = 0; i < limit; i+=2){
    acc = (acc OP data[i]) OP data[i+1];
  }
  for( ; i < size; i++){
    acc = acc OP data[i];
  }
  *tot = acc;
}
```

combine5.c
(Unrolled 2x w/ 1 Accumulator)

```
void combine7(struct vec* restrict v1, DTYPE*
restrict tot)
{
  *tot = IDENT;
  unsigned size = get_size(v1);
  DTYPE* data = v1->dat;
  DTYPE acc = IDENT;
  unsigned limit = size-1;
  unsigned i;
  for(i = 0; i < limit; i+=2){
    acc = acc OP (data[i] OP data[i+1]);
  }
  for( ; i < size; i++){
    acc = acc OP data[i];
  }
  *tot = acc;
}
```

combine7.c
(Unrolled 2x w/ 1A Accumulator)

Function	Method	Int (+)	Int (*)	FP (+)	FP (*)
combine5	Unrolled 2x1	1.01	3.01	3.01	5.01
combine7	Unrolled 2x1A	1.01	1.51	1.51	2.51

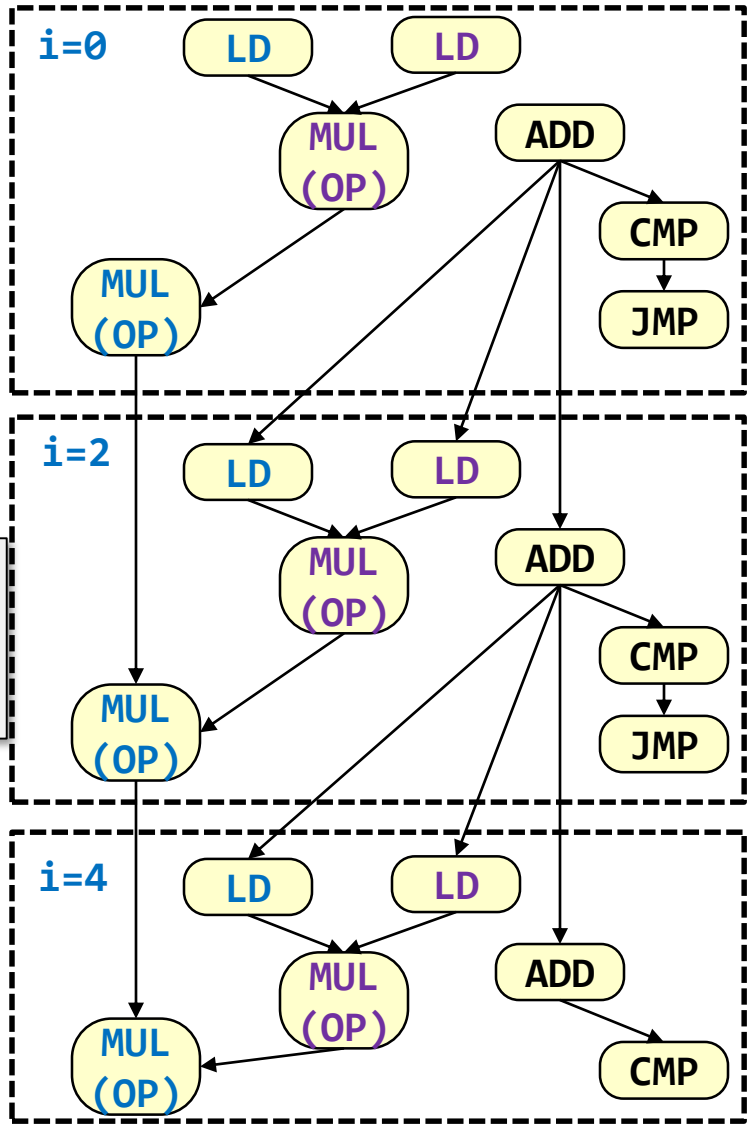
Combine7 Dataflow

```
void combine7(struct vec* restrict v1, DTYPE*
restrict tot)
{
  *tot = IDENT;
  unsigned size = get_size(v1);
  DTYPE* data = v1->dat;
  DTYPE acc = IDENT;
  unsigned limit = size-1;
  unsigned i;
  for(i = 0; i < limit; i+=2){
    acc = acc OP (data[i] OP data[i+1]);
  }
  for( ; i < size; i++){
    acc = acc OP data[i];
  }
  *tot = acc;
}
```

```
.L3:
        ld      (%rcx), %xmm1
        ld      8(%rcx), %xmm2
        mulsd   %xmm2, %xmm1
        mulsd   %xmm1, %xmm0
        addl    $16, %rcx
        cmpl    %edx, %edi
        ja      .L3
```

Critical path shortened to n/2 multiplies.

n/2 iters



Combine8 (Loop Unrolling)

- Further unrolling can also help achieve near-ideal (maximum) throughput

```
void combine8(struct vec* restrict v1, DTYPE*
restrict tot)
{
  *tot = IDENT;
  unsigned size = get_size(v1);
  DTYPE* data = v1->dat;
  DTYPE acc = IDENT;
  unsigned limit = size-9;
  unsigned i;
  for(i = 0; i < limit; i+=10){
    acc = acc OP ((data[i] OP data[i+1]) OP
                  (data[i+2] OP data[i+3]) OP
                  ...
                  (data[i+8] OP data[i+9]));
  }
  for( ; i < size; i++){
    acc = acc OP data[i];
  }
  *tot = acc;
}
```

combine8.c
(Unrolled 10x w/ 10 accumulators)

Function	Method	Int (+)	Int (*)	FP (+)	FP (*)
combine8	Unroll 10x10	0.55	1.00	1.01	0.52
Ideal	Latency:1/Throughput	1 : 0.5	3 : 1	3 : 1	5 : 0.5

Summary

- Compiler can perform some optimizations and even some unrolling, especially at higher levels of optimization (i.e. gcc/g++ -O3)
- Use a profiler to find the bottleneck
- Some manual transformations can help
  - Explicit unrolling
  - Code motion (factoring code out of a loop)
  - Avoid memory aliasing

Vector units

SIMD

Introductory Example for SIMD

- An image is just a 2D array of pixel values
- Pixel color represented as a numbers
  - E.g. 0 = black, 255 = white

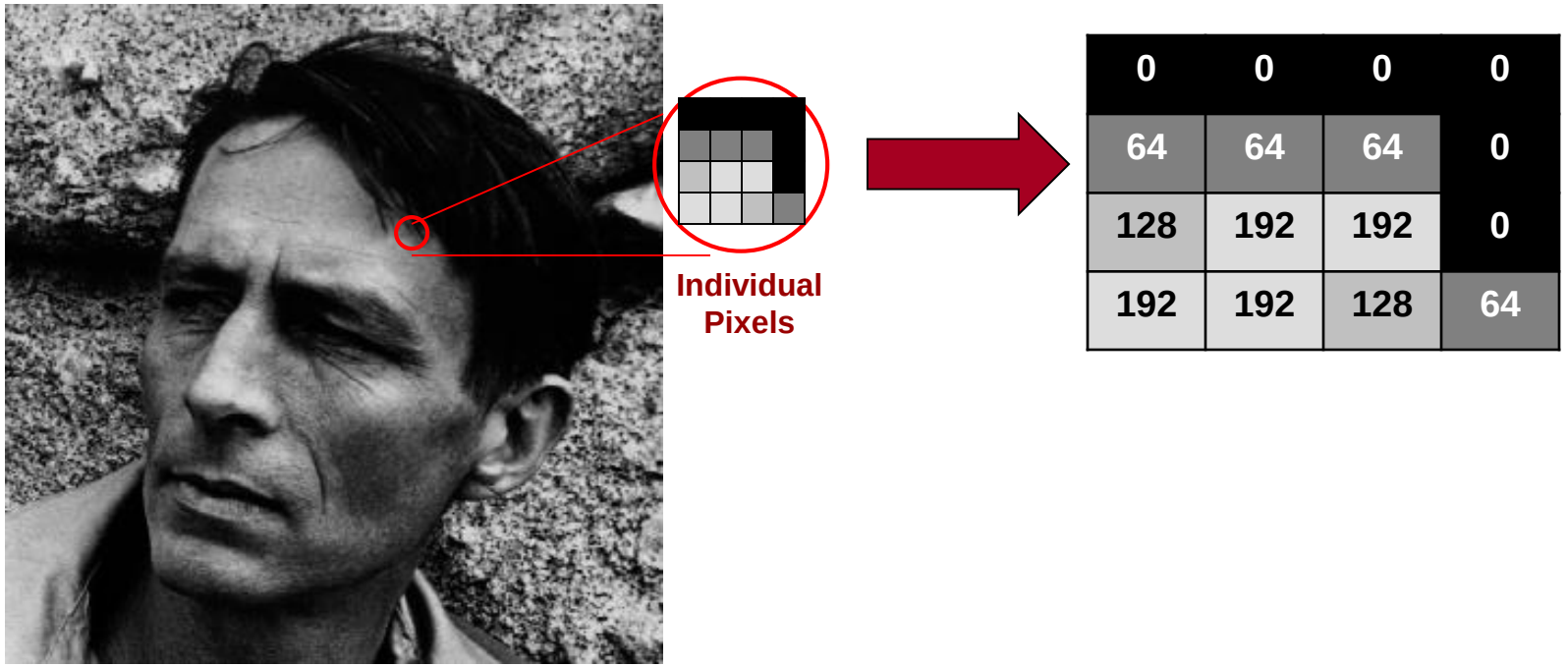


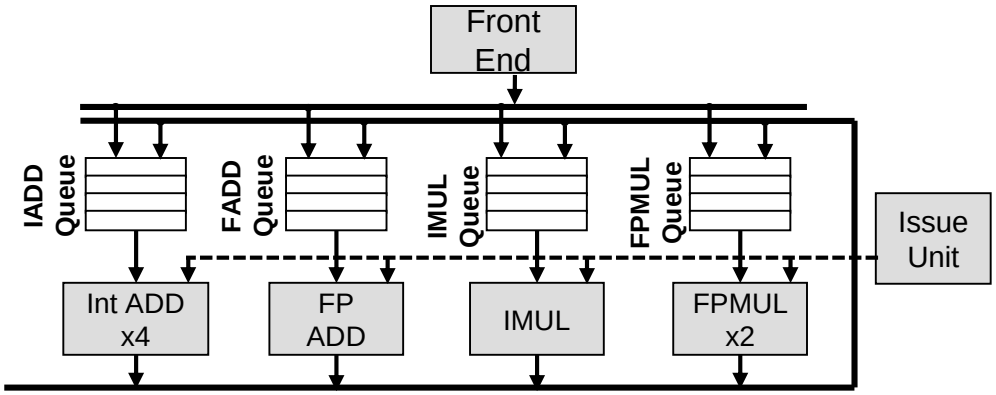
Image taken from the photo "Robin Jeffers at Ton House" (1927) by Edward Weston

Graphics Operations

- Brightness
  - Each pixel value is increased/decreased by a constant amount
  - $P_{new} = P_{old} + B$
    - B > 0 = brighter
    - B < 0 = less bright
- Contrast
  - Each pixel value is multiplied by a constant amount
  - $P_{new} = C*P_{old}$
    - C > 1 = more contrast
    - 0 < C < 1 = less contrast
- Same operations performed on all pixels

Scalar Operations

- Typical processors use instructions that perform an operation on a single (aka scalar) data value:
- Scalar: 1 instruc. = 1 data result
  - `movq (%rdi), %rax`
  - `addq %rdx, %rax`
- Referred to as SISD operation
  - Single Instruction, Single Data



1:1 ratio of instruction to data results

```
void combine4(struct vec* v1, DTYPE* tot)
{
  *tot = IDENT;
  unsigned size = get_size(v1);
  DTYPE* data = v1->dat;
  DTYPE acc = IDENT;
  for(unsigned i = 0; i < size; i++){
    acc = acc OP data[i];
  }
  *tot = acc;
}
```

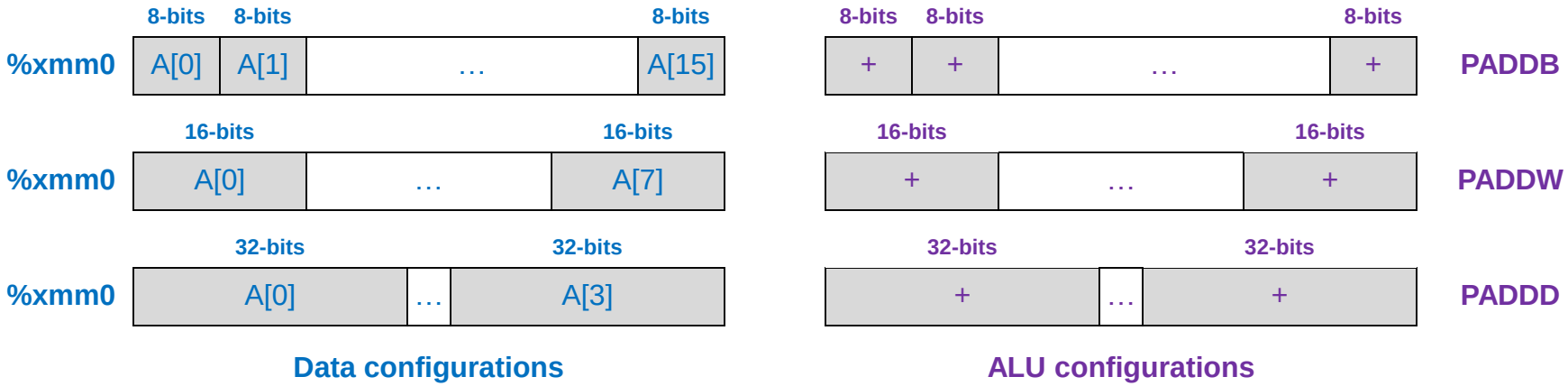
```
.L3:
        mulsd   (%rdx), %xmm0
        addq    $8, %rdx
        cmpq    %rax, %rdx
        jne     .L3
```

Vector/SIMD Operations

- Modern processors now include additional hardware that perform the same operation on multiple data items (a.k.a. vectors) using a single instruction
  - Referred to as SIMD (Single Instruction, Multiple Data) operation or vector or streaming operations
- Updated hardware capabilities include:
  - Processor adds vector registers which can each hold 128-, 256-, or even 512-bits of data
  - The 128-, 256-, or 512-bit data can be then interpreted as a **packed** set of 8, 16, or 32-data items and in a single instruction perform an operation on all of the packed data items.
  - Example instruction: `paddw (%rdi),%xmm0` where paddd (packed add dword) reads 128-bits of data (from cache) which is really (4) 32-bit dwords and adds it to another (4) 32-bit dwords in the 128-bit %xmm0 register

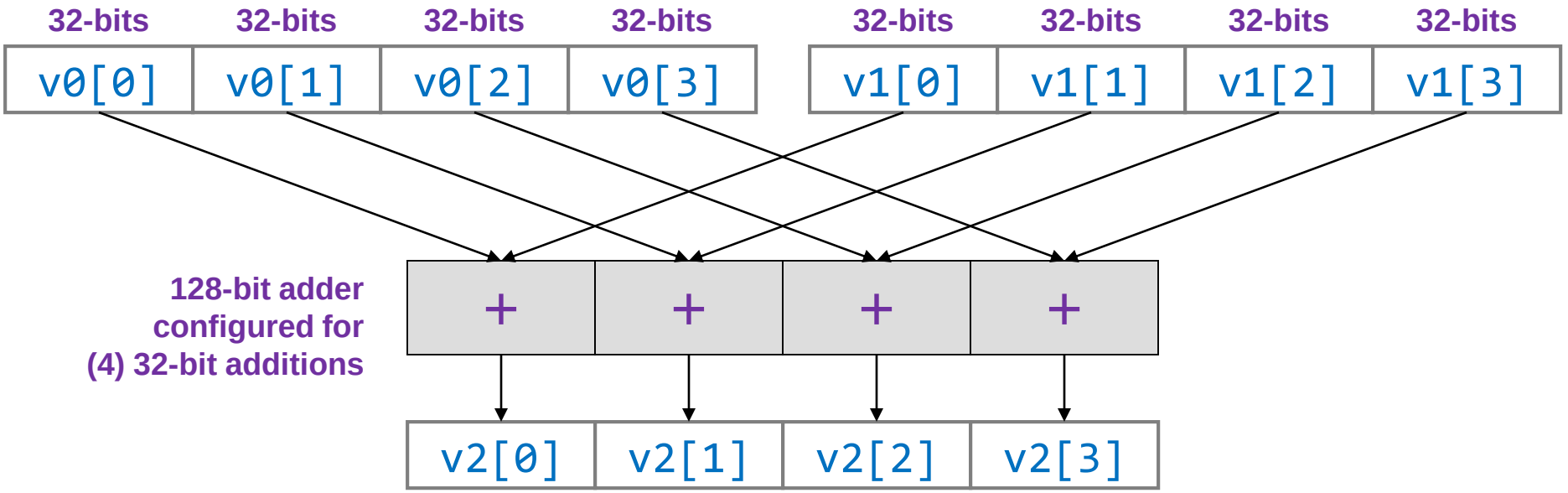
Vector Instructions

- Vector load / store
  - MOVDQA (Move Double Quad Word Aligned) reads a 128-bit chunk from memory to 128-bit vector register
  - `movdqa (%rdi), %xmm1`
- Vector operations (packed) [e.g. `paddb $5, %xmm0`]
  - PADDB = (16) 8-bit values
  - PADDW = (8) 16-bit values
  - PADDD = (4) 32-bit values



Vector Operations

- `paddd %xmm0, %xmm1`



More Vector Instructions

```
void f1(int* A, int n) {
  for( ; n != 0; n--, A++)
    *A += 5;
}
```

Original "scalar" code

```
// Loop unrolled 4 times
for(i=0; i < MAX; i+=4){
  A[i] = A[i] + 5;
  A[i+1] = A[i+1] + 5;
  A[i+2] = A[i+2] + 5;
  A[i+3] = A[i+3] + 5;
}
```

Unrolled Scalar Code

```
# %rdi = A
# %esi = n = # of iterations
L1: ld   0(%rdi),%r9
    add  $5,%r9
    st   %r9,0(%rdi)
    ld   4(%rdi),%r9
    add  $5,%r9
    st   %r9,4(%rdi)
    ld   8(%rdi),%r9
    add  $5,%r9
    st   %r9,8(%rdi)
    ld   12(%rdi),%r9
    add  $5,%r9
    st   %r9,12(%rdi)
    add  $16,%rdi
    add  $-4,%esi
    jne  $0,%esi,L1
```

Vectorized / SIMD Code (Could unroll this if desired)

```
# %rdi = A
# %esi = n = # of iterations

   .align 16
.LC1:
   .long   5,5,5,5
   ...
f1:
   movdqa  .LC1(%rip), %xmm0
L1:
   movdqa (%rdi), %xmm1
   paddd  %xmm0, %xmm1
   movdqa %xmm1,(%rdi)
   add    $16,%rdi
   addi   $-4,%esi
   jne    $0,%esi,L1
```

Vector Processing Examples

- Intel
  - SSE,SSE2,SSE3 – Streaming SIMD Extensions
    - 128-bit vectors & registers (%xmm0-%xmm15)
    - Support for (16) 8-bit, (8) 16-bit, or (4) 32-bit integers or (4) single- and (2) double-precision FP ops
  - AVX – Advanced Vector Extensions
    - 256-bit vectors (%ymm0-%ymm15)
    - Support for (32) 8-bit, (16) 16-bit, or (8) 32-bit integers or (8) single- and (4) double-precision FP ops
- ARM SVE (Scalable Vector Extensions)

Function	Method	Int (+)	Int (*)	FP (+)	FP (*)
combine8	Scalar 10x10	0.54	1.01	1.01	0.52
Ideal ScalarLatency:1/Throughput		1 : 0.5	3 : 1	3 : 1	5 : 0.5
SIMD	Vector 8x8	0.05	0.24	.25	0.16

Enabling Vectorization

Reference:
http://hpac.rwth-aachen.de/teaching/sem-accg-16/slides/08.Schmitz-GGC_Autovec.pdf

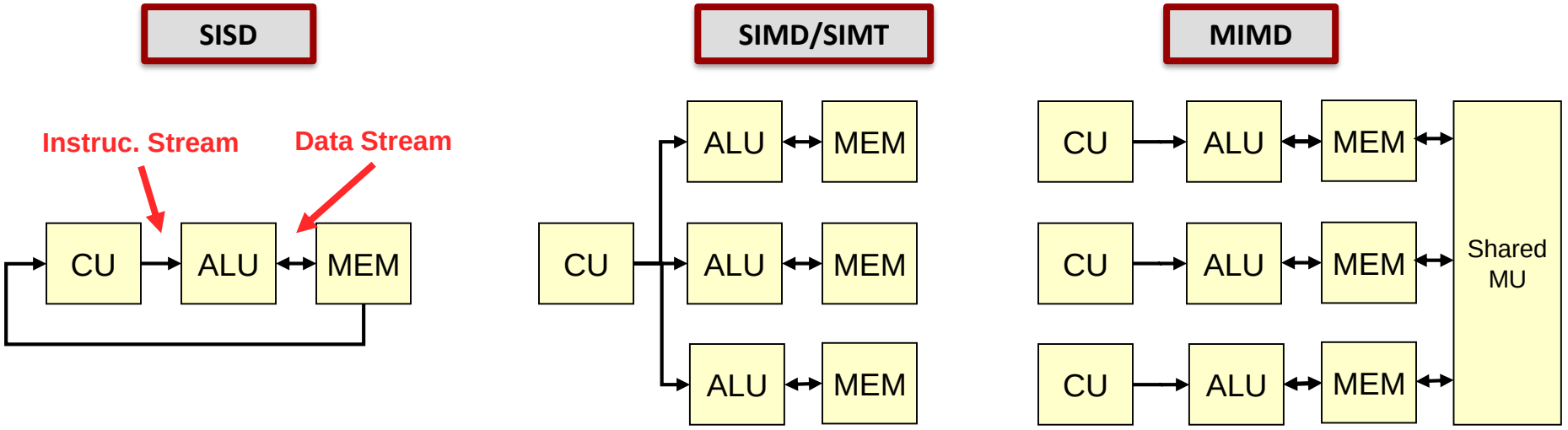
- To enable vectorization:
  - Use at least `-O3` with `gcc/g++`
  - Need to ensure memory alignment of arrays
    - Chunks of 16-bytes needs to start on an address that is a multiple of 16
  - Avoid memory aliasing
    - `restrict` keyword can help
- g++ options
  - `-fopt-info-vec`
    - See report of what loops were vectorized
  - `-march=native`
    - use native processor's capabilities

```
// simd1.c
void vec2(int* restrict A, unsigned n)
{
  A = (int*)
        __builtin_assume_aligned(A,32);

  for(unsigned i = 0; i < n; i++){
    A[i] += 5;
  }
}
```

```
$ gcc -O3 -march=native -fopt-info-vec -S simd1.c

simd1.c:13:3: note: loop vectorized
```

Parallel Processing Paradigms

- SISD = Single Instruction, Single Data
  - Uniprocessor
- SIMD = Single Instruction, Multiple Data/Thread
  - Multimedia/Vector Instruction Extensions, Graphics Processor Units (GPU's)
- MIMD = Multiple Instruction, Multiple Data
  - Typical multicore, parallel processing system



SIMT Example: NVIDIA Tesla GPU