1 Performance Optimization

1.1 Performance of arithmetic operations on the reference machine

Table 1: Performance & bound of arithmetic operations

Operation	Integer			Floating Number		
	Latency	Issue	Capacity	Latency	Issue	Capacity
Addition	1	1	4	3	1	1
Multiplication	3	1	1	5	1	2
Division	3-30	3-30	1	3-15	3-15	1
Bound	Integer			Floating Number		
	+	*		+	*	
Latency	1.00	3.00		3.00	5.00	
Throughput	0.50	1.00		1.00	0.50	

1.2 Techniques

Various techniques for performance optimization are illustrated on the following code.

```
#define IDENT 1 //#define IDENT 0
#define OP *    //#define OP +
typedef struct {
  long len;
  data_t *data;
} vec_rec, *vec_ptr;

int get_vec_element(vec_ptr v, long index, data_t *dest) {
  if(index < 0 || index > v->len) return 0;
  *dest = v->data[index];
  return 1;
}

long vec_length(vec_ptr v) { return v->len; }

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

1.2.1 Code motion

```
void combine2(vec_ptr v, data_t *dest) {
  long i;
  long length = vec_length(v);
  *dest = IDENT;
  for(i = 0; i < length; ++i) {
    data_t val;
    get_vec_element(v, i, &val);
    *dest = *dest OP val;
  }
}
```

1.2.2 Reduce procedure calls

```
data_t *get_vec_start(vec_ptr v) { return v->data; }

void combine3(vec_ptr v, data_t *dest) {
  long i;
  long length = vec_length(v);
  data_t *data = get_vec_start(v);
  *dest = IDENT;
  for(i = 0; i < length; ++i) {
    *dest = *dest OP data[i];
  }
}
```

1.2.3 Eliminate unnecessary memory references

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

1.2.4 Loop unrolling (2 * 1 unrolling)

```
void combine5(vec_ptr v, data_t *dest) {
  long i;
  long length = vec_length(v);
  data_t *data = get_vec_start(v);
  data_t acc = IDENT;
  long limit = length - 1;//length - k + 1 for k
  for(i = 0; i < limit; i+=2) { //i+=k for k
    acc = (acc OP data[i]) OP data[i + 1];
  }
  for(; i < length; ++i) acc = acc OP data[i];
  *dest = acc;
}
```

1.2.5 Multiple accumulator (2 * 2 unrolling)

```
void combine6(vec_ptr v, data_t *dest) {
  long i;
  long length = vec_length(v);
```

```
  data_t *data = get_vec_start(v);
  long limit = length - 1;//length - k + 1 for k
  data_t acc0 = IDENT;
  data_t acc1 = IDENT;
  for(i = 0; i < limit; i+=2) { //i+=k for k
    acc0 = acc0 OP data[i];
    acc1 = acc1 OP data[i + 1];
  }
  for(; i < length; ++i) acc0 = acc0 OP data[i];
  *dest = acc0 OP acc1;
}
```

1.2.6 Reassociation (2 * 1a unrolling)

```
void combine5(vec_ptr v, data_t *dest) {
  long i;
  long length = vec_length(v);
  data_t *data = get_vec_start(v);
  data_t acc = IDENT;
  long limit = length - 1;//length - k + 1 for k
  for(i = 0; i < limit; i+=2) { //i+=k for k
    acc = acc OP (data[i]) OP data[i + 1];
  }
  for(; i < length; ++i) acc = acc OP data[i];
  *dest = acc;
}
```

1.3 Limitations

1. Register spilling. When the degree of parallelism p exceeds the number of registers, some temporaries will have to be stored on stack, which harms the performance of the program.

2. Penalty of wrong branch prediction. Usually we do not have to worry too much about the penalty. But we should write code suitable for conditional moves when possible.

Imperative Code	Functional Code

`if(a[i] > b[i]) {`
`  long t = a[i];`
`  a[i] = b[i];`
`  b[i] = t;`
`}`
	`long min = a[i] < b[i] ? : a[i] : b[i];`
	`long max = a[i] < b[i] ? b[i] : a[i];`
	`a[i] = min;`
	`b[i] = max;`

`if(src1[i1] < src2[i2])`
`  dest[id++] = src1[i1++];`
`else`
`  dest[id++] = src2[i2++];`
	`long v1 = src[i1], v2 = src[i2];`
	`bool choose1 = v1 < v2;`
	`dest[id++] = choose1 ? v1 : v2;`
	`i1 += choose1;`
	`i2 += (1 - choose1);`

1.4 Memory Performance

Try to avoid loading value from a memory position immediately after writing to it, which increases the length of the critical path. For example, this function:

```
void psum1(float a[], float p[], long n) {
  long i;
  p[0] = a[0];
  for(i = 1; i < n; i++) {
    p[i] = p[i - 1] + a[i];//Load from p[i-1], which was written in the last
        iteration.
  }
}
```

should be written as:

```
void psum1_better(float a[], float p[], long n) {
  long i;
  float val = a[0];
  p[0] = val;
  for(i = 1; i < n; i++) {
    val += a[i];
    p[i] = val;//val is in register, not in memory.
  }
}
```