Optimization

HGU

Code Optimization

- **Optimization** is the process by which a program is improved by reducing its code size, complexity, memory use, or runtime (or some combination thereof) without changing the program's inherent function

- GCC compiler supports various optimization flags that allow programmers to directly invoke different **optimization levels**
  - **-O*n* (capital O and 1,2, or 3)**

```
$ gcc -O1 -o program program.c
```

GCC Optimization Levels

- `-O0` is the default which make debugging produce
- `-O1` tries to reduce code size and execution time
  ex) Dead Code Elimination, Simple Loop Optimizations, etc.
- `-O2` performs nearly all optimizations in a space-speed tradeoff
  ex) Inline Functioning, Loop Unrolling, Copy Propagation, etc.
- `-O3` turns on all optimizations
  ex) Loop Transformations, Software Pipelining, etc.
- The other optimization flags exist: `-Og`, `-Os`, `-Ofast`, …

Optimizations

- What compilers already do :
  - Constant Folding
  - Constant Propagation
  - Dead Code Elimination
  - Simplifying Expressions

Optimizations

- What compilers already do :

<table>
<tr><th>Original Version (tripleSum.c)</th><th>Optimized Version</th></tr>
<tr><td>

```c
#define N 5
int debug = N - 5;

int tripleSum(int* array, int length) {
    int i, total = 0;
    for (i = 0; i < length; i++) {
        total += array[i];

        if (debug)
           printf("%d:%d\n",i,array[i]);
    }

    return 3 * total;
}
```

</td><td>

```c
//Constant Folding
int debug = 0;

int tripleSum(int* array, int length) {
    int i, total = 0;
    for (i = 0; i < length; i++) {
        total += array[i];

        if (debug)
           printf("%d:%d\n",i,array[i]);
    }

    return 3 * total;
}
```

</td></tr>
</table>

Optimizations

- What compilers already do :

Original Version (tripleSum.c)	Optimized Version

`#define N 5`
`int debug = N - 5;`
 	
`int tripleSum(int* array, int length) {`
`    int i, total = 0;`
`    for (i = 0; i < length; i++) {`
`        total += array[i];`
 	
`        if (debug)`
`            printf("%d:%d\n",i,array[i]);`
`    }`
 	
`    return 3 * total;`
`}`
	`//Constant Folding`
	`int debug = 0;`
	
	`int tripleSum(int *array, int length){`
	`    int i, total = 0;`
	`    for (i = 0; i < length; i++){`
	`        total += array[i];`
	
	`        if (0) //Constant Propagation`
	`            printf("%d:%d\n",i,array[i]);`
	`    }`
	
	`    return 3 * total;`
	`}`

Optimizations

- What compilers already do :

Original Version (tripleSum.c)	Optimized Version

<pre>#define N 5
int debug = N - 5;
 	
int tripleSum(int* array, int length) {
    int i, total = 0;
    for (i = 0; i < length; i++) {
        total += array[i];
 	
        if (debug)
           printf("%d:%d\n",i,array[i]);
    }
 	
    return 3 * total;
}</pre>
	<pre>//Constant Folding
	int debug = 0;
	
	int tripleSum(int *array, int length){
	    int i, total = 0;
	    for (i = 0; i < length; i++){
	        total += array[i];
	//Dead Code Elimination
	// if (0) //Constant Propagation
	// printf("%d:%d\n",i,array[i]);
	    }
	
	    return 3 * total;
	}</pre>

Optimizations

- What compilers already do :

Original Version (tripleSum.c)	Optimized Version

<pre>#define N 5
int debug = N - 5;
 	
int tripleSum(int* array, int length) {
    int i, total = 0;
    for (i = 0; i < length; i++) {
        total += array[i];
 	
        if (debug)
           printf("%d:%d\n",i,array[i]);
    }
 	
    return 3 * total;
}</pre>
	<pre>//Constant Folding
	int debug = 0;
	
	int tripleSum(int *array, int length){
	    int i, total = 0;
	    for (i = 0; i < length; i++){
	        total += array[i];
	//Dead Code Elimination
	// if (0) //Constant Propagation
	// printf("%d:%d\n",i,array[i]);
	    }
	    //Simplifying Expressions
	    return (total << 2) + total;
	}</pre>

What compiler can not always do

- **Algorithmic strength reduction is Impossible: Bad choices of data** structures and algorithms cannot be fixed by compiler

- **Compiler Optimization flags are not guaranteed to make code optimal**: Sometimes, increasing optimization Level (e.g. from –O2 to –O3) may slows down a program. In other cases, compiling with –O2 or –O3 results in segmentation fault while compiling without optimization flags to run without errors.

- **Pointers can prove Problematic**: Due to Memory aliasing problem (two different pointers point to the same memory location), compilers do not make transformation if there are risks of program behavior changes.

Compiler Optimization with Undefined Behaviors

- The compiler is not required to handle undefined behavior (inconsistent output is not a flaw of compiler)

<table>
<tr><th>silly.c</th><th>Execution Command and Result</th></tr>
<tr><td rowspan="4"><pre>#include <stdio.h>
#include <limits.h>

int silly(int x) {
  int tmp = x+1;
  printf("%d, %x\n", tmp, tmp);
  printf("%d, %x\n", x, x);
  return tmp > x;
}

int main() {
        int max = INT_MAX;
       printf("%d\n", silly(max));
}</pre></td><td><code>$ gcc -O3 -o silly_opt silly.c && ./silly_opt</code></td></tr>
<tr><td>-2147483648, 80000000
2147483647, 7fffffff
1</td></tr>
<tr><td><code>$ gcc -o silly silly.c && ./silly</code></td></tr>
<tr><td>-2147483648, 80000000
2147483647, 7fffffff
0</td></tr>
</table>

No Transformation with Optimization Flags

- Example of problem caused by Memory Aliasing

<table>
<tr><th>Original Version (shiftadd.c)</th><th>Optimized Version</th></tr>
<tr><td>

```c
void shiftAdd(int *a, int *b){
    *a = *a * 10; //multiply by 10
    *a += *b;     //add b
}

int main() {
    int x = 5, y = 6;
    shiftAdd(&x, &y);
    printf("shiftAdd : %d\n", x);
    x = 5;
    shiftAdd(&x, &x);
    printf("shiftAdd : %d\n", x);
    return 0;
}
```

</td><td>

```c
void shiftAddOpt(int *a, int *b){
    // Reducing a memory reference
    *a = (*a * 10) + *b;
}

int main() {
    int x = 5, y = 6;
    shiftAdd(&x, &y);
    printf("shiftAddOpt : %d\n", x);
    x = 5;
    shiftAddOpt(&x, &x);
    printf("shiftAddOpt : %d\n", x);
    return 0;
}
```

</td></tr>
<tr><td>

```
shiftAdd : 56
shiftAdd : 100
```

</td><td>

```
shiftAddOpt : 56
shiftAddOpt : 55
```

</td></tr>
</table>

Time Measurement Functions (1)

- clock_gettime() uses two timer
  - CLOCK_REALTIME : system real time
  - CLOCK_MONOTONIC : the time after system boots

Example of clock_gettime()

```c
#include <time.h>

int main() {
  struct timespec start, end;
  clock_gettime(CLOCK_MONOTONIC, &start);
  ...
  clock_gettime(CLOCK_MONOTONIC, &end);

  double elapsed = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;
  printf("Elapsed time: %.9f seconds\n", elapsed);
}
```

Time Measurement Functions (2)

- gettimeofday() returns the current system time as second or microseconds

Example of gettimeofday()

```
#include <sys/time.h>

int main() {
  struct timeval start, end;
  gettimeofday(&start, NULL);
  ...
  gettimeofday(&end, NULL);

  double elapsed = (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) / 1e6;
  printf("Elapsed time: %.6f seconds\n", elapsed);
}
```

Optimization Skill Example

Source Code (genPrime.c)

```c
//helper function:
//checks to see if a number is prime
int isPrime(int x) {
    int i;
    //no prime number is less than 2
    for (i = 2; i < sqrt(x) + 1; i++)
        if (x % i == 0)
            return 0;
    return 1;
}
// finds the next prime
int getNextPrime(int prev) {
    int next = prev + 1;
    while (!isPrime(next))
        next++;
    return next;
}
```

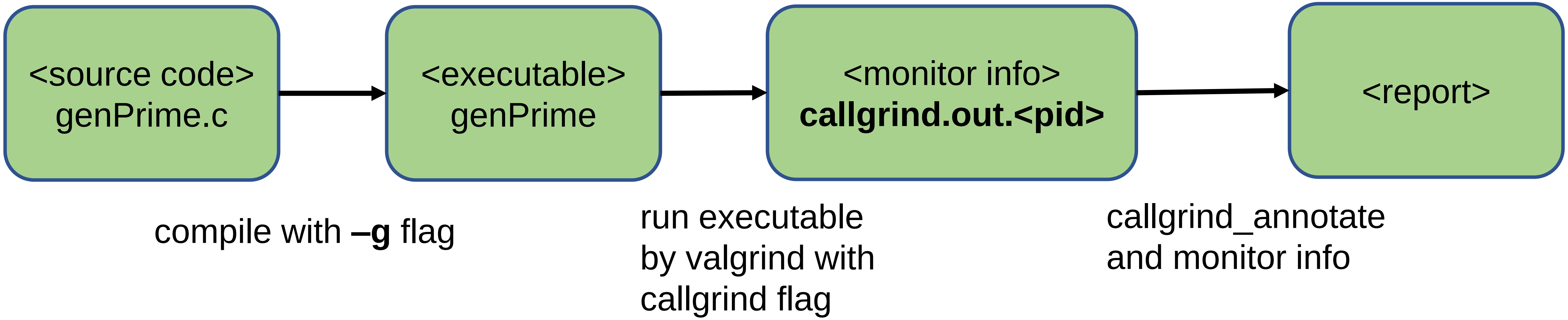
```c
// generates a sequence of primes
int genPrimeSequence(int *array, int limit) {
    int i;
    int len = limit;
    if (len == 0) return 0;
    array[0] = 2;
    for (i = 1; i < len; i++) {
        array[i] = getNextPrime(array[i-1]);
        if (array[i] > limit) {
            len = i;
            return len;
        }
    }
    return len;
}
int main(int argc, char **argv) {
  //omitted for brevity
  int *array = allocateArray(limit);
  int length = genPrimeSequence(array, limit);
  free(array);
  return 0;
}
```

Optimization with Profiling

- Profiling : a process in software development that involves measuring the performance of a program or application.
- Objective: to identify which parts of the code are consuming the most resources, such as CPU time, memory, or I/O operations. By analyzing this data, developers can optimize the program's performance by focusing on the most critical bottlenecks.
- gprof
  - GNU Project profiler
  - gcc –pg –o myprog myprog.c
  - myprog
  - gprof myprog gmon.out >analysis.txt
- valgrind (callgrind)
  - gcc –g –o myprog myprog.c
  - valgrind –tool=callgrind myprog
  - callgrind_annotate callgrind.out.<pid>
- perf
  - gcc –o myprog myprog.c
  - perf record –g myprog
  - perf report

Profiling Process with valgrind

- $ gcc **–g** genPrime.c –o genPrime –lm
- $ **valgrind --tool=callgrind** genPrime 10000000
- $ callgrind_annotate --auto=yes **callgrind.out.*<pid>***



Optimizations with Valgrind

- Using **Callgrind** to Profile

```
$ gcc -g -o genPrime genPrime.c -lm
$ valgrind --tool=callgrind ./genPrime 100000
```

```
==3709== Callgrind, a call-graph generating cache profiler
==3709== Copyright (C) 2002-2015, and GNU GPL'd, by Josef Weidendorfer et al.
==3709== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==3709== Command: ./genPrime 100000
==3709==
==3709== For interactive control, run 'callgrind_control -h'.
Time to generate primes: 0.16056
9592 primes found.
==3709==
==3709== Events    : Ir
==3709== Collected : 28278690
==3709==
==3709== I   refs:      28,278,690
```

callgrind.out.3709
file is generated

```
$ callgrind_annotate --auto=yes callgrind.out.3709
```

Optimizations with Valgrind

- Using **Callgrind** to Profile

$ callgrind_annotate --auto=yes callgrind.out.3709

```
          .  //helper function: checks to see if a number is prime
    400,004  int isPrime(int x) {
          .      int i;
 36,047,657      for (i = 2; i < sqrt(x)+1; i++) { //no prime number is less than 2
 13,826,015  => /build/glibc-S7Ft5T/glibc-2.23/math/w_sqrt.c:sqrt (2765203x)
        796  => /build/glibc-S7Ft5T/glibc-2.23/elf/../sysdeps/x86_64/dl-trampoline.h:_dl_runtime_resolve_xsave'2 (1x)
 16,533,672          if (x % i == 0) { //if the number is divisible by i
    180,818              return 0; //it is not prime
          .          }
          .      }
      9,592      return 1; //otherwise it is prime
    200,002  }
          .
          .  // finds the next prime
     38,368  int getNextPrime(int prev) {
     28,776      int next = prev + 1;
    509,597      while (!isPrime(next)) { //while the number is not prime
 67,198,556  => genPrime.c:isPrime (100001x)
     90,409          next++; //increment and check again
          .      }
      9,592      return next;
     19,184  }
```

Optimizations : Loop-Invariant Code Motion

- Compare the original and optimized version

Original Version (genPrime.c)	Optimized Version (genPrime1.c)

`//checks to see if a number is prime`
`int isPrime(int x) {`
`    int i;`
 	
`    for (i = 2; i < sqrt(x) + 1; i++)`
`        if (x % i == 0)`
`            return 0;`
`    return 1;`
`}`
	`int isPrime(int x) {`
	`    int i;`
	`    //Loop-Invariant Code Motion`
	`    int max = sqrt(x)+1;`
	`    for (i = 2; i < max; i++)`
	`        if (x % i == 0)`
	`            return 0;`
	`    return 1;`
	`}`

GCC Command	Unoptimized	-O1	-O2	-O3
`$ gcc -o original genPrime.c -lm`	3.48	2.32	2.14	2.15
`$ gcc -o loop-invariant genPrime1.c -lm`	1.83	1.63	1.71	1.63

Optimizations with Valgrind

- Using **Callgrind** to Profile

```
$ gcc -g -o loop-invariant genPrime.c -lm
$ valgrind --tool=callgrind ./loop-invariant 100000
$ callgrind_annotate --auto=yes callgrind.out.18125
```

```
         .  //helper function: checks to see if a number is prime
   400,004  int isPrime(int x) {
         .      int i;
         .      //Loop-Invariant Code Motion
   900,013      int max = sqrt(x)+1;
       796  => /build/glibc-S7Ft5T/glibc-2.23/elf/../sysdeps/x86_64/dl-trampoline.h:_dl_runtime_resolve_xsave'2 (1x)
   500,000  => /build/glibc-S7Ft5T/glibc-2.23/math/w_sqrt.c:sqrt (100000x)
11,122,449      for (i = 2; i < max; i++) {
16,476,120          if (x % i == 0) {
   180,818          return 0; //it is not prime
         .          }
         .      }
     9,592      return 1; //otherwise it is prime
   200,002  }
         .
         .  // finds the next prime
    38,368  int getNextPrime(int prev) {
    28,776      int next = prev + 1;
   509,597      while (!isPrime(next)) { //while the number is not prime
29,789,794  => genPrime.c:isPrime (100001x)
    90,409          next++; //increment and check again
         .      }
     9,592      return next;
    19,184  }
```

Optimizations : Loop Unrolling (1)

- Compare the original and optimized version

Original Version (genPrime1.c)	Optimized Version (genPrime2.c)

<pre>int isPrime(int x) {
    int i;
    int max = sqrt(x)+1;
 	
    for (i = 2; i < max; i++)
        if (x % i == 0)
            return 0;
    return 1;
}</pre>
	<pre>int isPrime(int x) {
	    int i;
	    int max = sqrt(x)+1;
	    //Loop Unrolling
	    for (i = 2; i < max; i+=2)
	      if ((x % i == 0) || (x % (i+1) == 0))
	          return 0;
	    return 1;
	}</pre>

GCC Command	Unoptimized	-O1	-O2	-O3
`$ gcc -o original genPrime.c -lm`	3.48	2.32	2.14	2.15
`$ gcc -o loop-invariant genPrime1.c -lm`	1.83	1.63	1.71	1.63
`$ gcc -o loop-unrolling genPrime2.c -lm`	1.65	1.53	1.45	1.45

Optimizations : Loop Unrolling (2)

- **Loop Unrolling**

  is a technique used to increase the execution speed by increasing the number of operations within the loop body, thus decreasing the number of iterations

Original Version (genPrime1.c)	Optimized Version (genPrime2.c)

<pre>int isPrime(int x) {
    int i;
    int max = sqrt(x)+1;
 	
    for (i = 2; i < max; i++)
        if (x % i == 0)
            return 0;
    return 1;
}</pre>
	<pre>int isPrime(int x) {
	    int i;
	    int max = sqrt(x)+1;
	    //Loop Unrolling
	    for (i = 2; i < max; i+=2)
	        if ((x % i == 0) || (x % (i+1) == 0))
	            return 0;
	    return 1;
	}</pre>

Compare Optimization Performance

- Compare with manual fix and optimization flags

GCC Command	Unoptimized	-O1	-O2	-O3
`$ gcc -o original genPrime.c -lm`	3.48	2.32	2.14	2.15
`$ gcc -o loop-invariant genPrime1.c -lm`	1.83	1.63	1.71	1.63
`$ gcc -o loop-invariant genPrime1.c -lm -O2 -funroll-loops`	1.82	1.48	1.46	1.46
`$ gcc -o loop-invariant genPrime1.c -lm -O2 -funroll-all-loops`	1.81	1.47	1.47	1.46
`$ gcc -o loop-unrolling genPrime2.c -lm`	1.65	1.53	1.45	1.45

Optimizations : Function Inlining

- **Function Inlining**

  replaces a function call with the actual code of the function to reduce the overhead associated with function calls

Original Version	Optimized Version

`int main(int argc, char **argv) {`
`    int lim = strtol(argv[1], NULL, 10);`
 	
`    // allocation of array`
`    int *a = allocateArray(lim);`
 	
`    int len = genPrimeSequence(a, lim);`
`    return 0;`
`}`
	`int main(int argc, char **argv) {`
	`    int lim = strtol(argv[1], NULL, 10);`
	
	`    // allocation of array (in-lined)`
	`    int *a = malloc(lim * sizeof(int));`
	
	`    int len = genPrimeSequence(a, lim);`
	`    return 0;`
	`}`

Optimizations Example #2

Source Code (matrixVector.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>

#define DEBUG 0

//helper function: computes wall clock time
double getTime(struct timeval ts, struct timeval te){
    double time = te.tv_sec - ts.tv_sec + (te.tv_usec - ts.tv_usec)/1.e6;
    return time;
}

//helper function: allocates an array of a specified length
int * allocateArray(int len) {
    int * result = malloc(len * sizeof(int));
    return result;
}
```

Optimizations Example #2

Source Code (matrixVector.c)

```
//helper function: fills array with elements
void fillArrayRandom(int * array, int len) {
    int i;
    for (i = 0; i < len; i++) {
        array[i] = 1+rand()%100;
    }
}

//helper function: fills an array with zeros
void fillArrayZeros(int * array, int len) {
    int i;
    for (i = 0; i < len; i++) {
        array[i] = 0;
    }
}

//helper function: prints out elements of array separated by spaces
void printArray(int * arr, int len) {
    long i;
    for (i = 0; i < len; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

Optimizations Example #2

Source Code (matrixVector.c)

```
//prints out the elements of a matrix, with each row on a separate line
void printMatrix(int ** mat, long rows, long cols) {
    long i;
    for (i = 0; i < rows; i++) {
        printArray(mat[i], cols);
    }
}

void matrixVectorMultiply(int ** mat, int * vec, int ** res, int row, int col){
    int i, j;
    for (j = 0; j < col; j++){
        for (i = 0; i < row; i++){
            res[i][j] = mat[i][j] * vec[j];
        }
    }
}

int main(int argc, char ** argv) {
    if (argc != 3) {
        fprintf(stderr, "usage: %s <n> <m>\n", argv[0]);
        printf("where <n> is the number of rows and <m> is the number of cols\n");
        printf("program will allocate a random nxm matrix and a vector of size m\n");
        printf("and perform matrix-vector multiplication on them.\n");
        return 1;
    }
```

Optimizations Example #2

Source Code (matrixVector.c)

```
struct timeval tstart, tend;
int rows = strtol(argv[1], NULL, 10);
int cols = strtol(argv[2], NULL, 10);
int i;
srand(4);

//declare, allocate and fill input and output matrices
gettimeofday(&tstart, NULL);
int ** matrix = malloc(rows*sizeof(int *));
int ** result = malloc(rows*sizeof(int *));

//allocate matrices
for (i = 0; i < rows; i++){
    matrix[i] = allocateArray(cols);
    result[i] = allocateArray(cols);
}

//fill matrices
for (i = 0; i < rows; i++){
    fillArrayRandom(matrix[i], cols);
    fillArrayZeros(result[i], cols);
}
gettimeofday(&tend, NULL);
printf("Time to allocate and fill matrices: %g\n", getTime(tstart, tend));
```

Optimizations Example #2

Source Code (matrixVector.c)

```
    //allocate and fill vector
    gettimeofday(&tstart, NULL);
    int * vector = allocateArray(cols);
    fillArrayRandom(vector, cols);
    gettimeofday(&tend, NULL);
    printf("Time to allocate vector: %g\n", getTime(tstart, tend));

    //perform matrix-vector multiplication
    gettimeofday(&tstart, NULL);
    matrixVectorMultiply(matrix, vector, result, rows, cols);
    gettimeofday(&tend, NULL);
    printf("Time to matrix-vector multiply: %g\n", getTime(tstart, tend));

    //print out matrix and result if the debug flag is on
    if (DEBUG) {
        printf("Matrix:\n");
        printMatrix(matrix, rows, cols);
        printf("\nVector:\n");
        printArray(vector, cols);
        printf("\nResult:\n");
        printMatrix(result, rows, cols);
    }
    return 0;
}
```

Optimizations with Valgrind

- Using **Callgrind** to Profile

```
$ gcc -g -o matrixVector matrixVector.c -lm
$ valgrind --tool=callgrind ./matrixVector 1000 1000
```

```
==4363== Callgrind, a call-graph generating cache profiler
==4363== Copyright (C) 2002-2015, and GNU GPL'd, by Josef Weidendorfer et al
==4363== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==4363== Command: ./matrixVector 1000 1000
==4363==
==4363== For interactive control, run 'callgrind_control -h'.
Time to allocate and fill matrices: 0.650406
Time to allocate vector: 0.000689
Time to matrix-vector multiply: 0.031994
==4363==
==4363== Events    : Ir
==4363== Collected : 115702601
==4363==
==4363== I   refs:      115,702,601
```

```
$ callgrind_annotate --auto=yes callgrind.out.4363
```

Optimizations with Valgrind

- Using **Callgrind** to Profile

```
$ callgrind_annotate --auto=yes callgrind.out.4363
```

```
         .        //fill matrices
     4,005        for (i = 0; i < rows; i++){
    10,000            fillArrayRandom(matrix[i], cols);
71,984,581  => matrixVector.c:fillArrayRandom (1000x)
    10,000            fillArrayZeros(result[i], cols);
10,012,000  => matrixVector.c:fillArrayZeros (1000x)
         .        }
```

```
         .        //perform matrix-vector multiplication
         5        gettimeofday(&tstart, NULL);
         5  => /build/glibc-S7Ft5T/glibc-2.23/time/../sysdeps/unix/sysv/linux/x86/gettimeofday.c:__gettimeofday_syscall (1x)
         8        matrixVectorMultiply(matrix, vector, result, rows, cols);
33,009,015  => matrixVector.c:matrixVectorMultiply (1x)
         5        gettimeofday(&tend, NULL);
         5  => /build/glibc-S7Ft5T/glibc-2.23/time/../sysdeps/unix/sysv/linux/x86/gettimeofday.c:__gettimeofday_syscall (1x)
        11        printf("Time to matrix-vector multiply: %g\n", getTime(tstart, tend));
     2,467  => /build/glibc-S7Ft5T/glibc-2.23/stdio-common/printf.c:printf (1x)
        30  => matrixVector.c:getTime (1x)
```

Optimizations : Loop Fission(1)

- Loop Fission

$ callgrind_annotate --auto=yes callgrind.out.680

```
          .        //fill matrices
      4,005        for (i = 0; i < rows; i++){
     10,000            fillArrayRandom(matrix[i], cols);
 71,984,581  => matrixVector.c:fillArrayRandom (1000x)
     10,000            fillArrayZeros(result[i], cols);
 10,012,000  => matrixVector.c:fillArrayZeros (1000x)
          .        }
```

Original Version (matrixVector.c)	Optimized Version (matrixVector1.c)

`for (i = 0; i < rows; i++) {`
`    fillArrayRandom(matrix[i], cols);`
`    fillArrayZeros(result[i], cols);`
`}`
	`//With Loop Fission`
	`for (i = 0; i < rows; i++) {`
	`    fillArrayRandom(matrix[i], cols);`
	`}`
	`for (i = 0; i < rows; i++) {`
	`    fillArrayZeros(result[i], cols);`
	`}`

Optimizations : Loop Fission(2)

- **Loop Fission**
  is an technique used to improve the performance by dividing a large loop into multiple smaller loops.

- Enhance Cache Performance
- Improve Parallelism
- Reduce Contention in Concurrent Environments

Original Version (matrixVector.c)	Optimized Version (matrixVector1.c)

`for (i = 0; i < rows; i++) {`
`    fillArrayRandom(matrix[i], cols);`
`    fillArrayZeros(result[i], cols);`
`}`
	`//With Loop Fission`
	`for (i = 0; i < rows; i++) {`
	`    fillArrayRandom(matrix[i], cols);`
	`}`
	`for (i = 0; i < rows; i++) {`
	`    fillArrayZeros(result[i], cols);`
	`}`

Optimizations : Loop Interchange(1)

- Loop Interchange

  $ callgrind_annotate --auto=yes callgrind.out.680

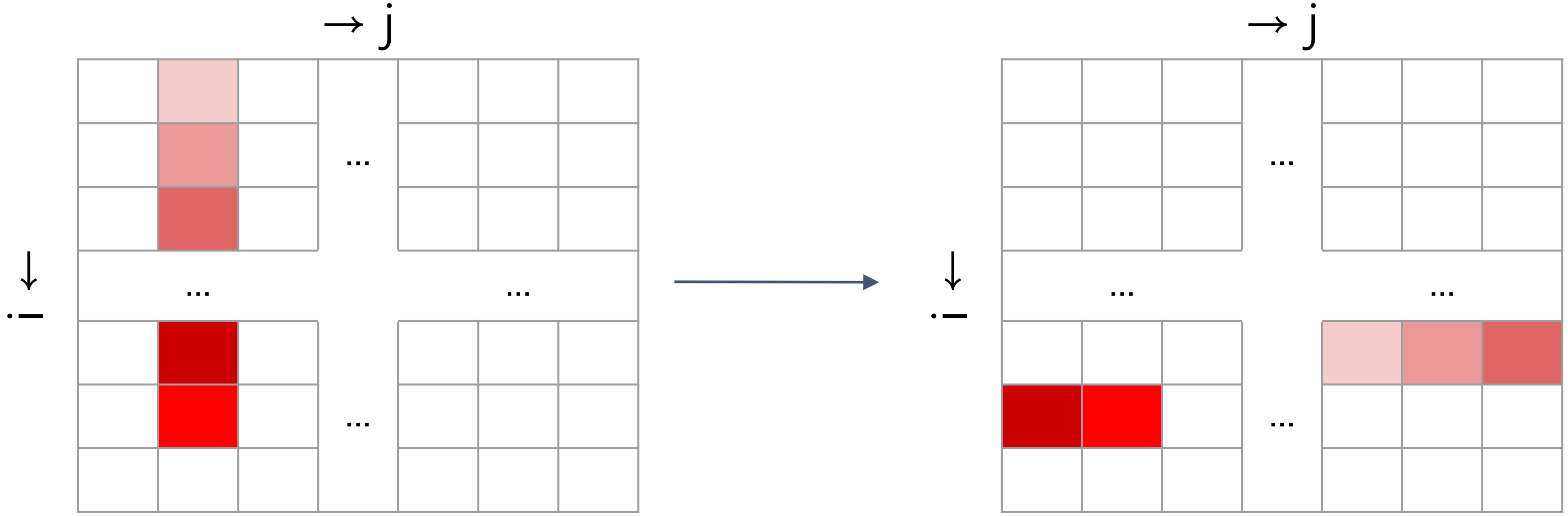
```
          .       //perform matrix-vector multiplication
          5       gettimeofday(&tstart, NULL);
          5   => /build/glibc-S7Ft5T/glibc-2.23/time/../sysdeps/unix/sysv/linux/x86/gettimeofday.c:__gettimeofday_syscall (1x)
          8       matrixVectorMultiply(matrix, vector, result, rows, cols);
 33,009,015   => matrixVector.c:matrixVectorMultiply (1x)
          5       gettimeofday(&tend, NULL);
          5   => /build/glibc-S7Ft5T/glibc-2.23/time/../sysdeps/unix/sysv/linux/x86/gettimeofday.c:__gettimeofday_syscall (1x)
         11       printf("Time to matrix-vector multiply: %g\n", getTime(tstart, tend));
      2,467   => /build/glibc-S7Ft5T/glibc-2.23/stdio-common/printf.c:printf (1x)
         30   => matrixVector.c:getTime (1x)
```

Original Version (matrixVector.c)	Optimized Version (matrixVector2.c)

void matrixVectorMultiply(int ** mat, int * vec, int ** res, int row, int col){
    int i, j;
 	
    for (j = 0; j < col; j++)
        for (i = 0; i < row; i++)
            res[i][j] = mat[i][j] * vec[j];
}
	void matrixVectorMultiply(int ** mat, int * vec, int ** res, int row, int col){
	    int i, j;
	    //Loop Interchange
	    for (i = 0; i < row; i++)
	        for (j = 0; j < col; j++)
	            res[i][j] = mat[i][j] * vec[j];
	}

Optimizations : Loop Interchange(2)

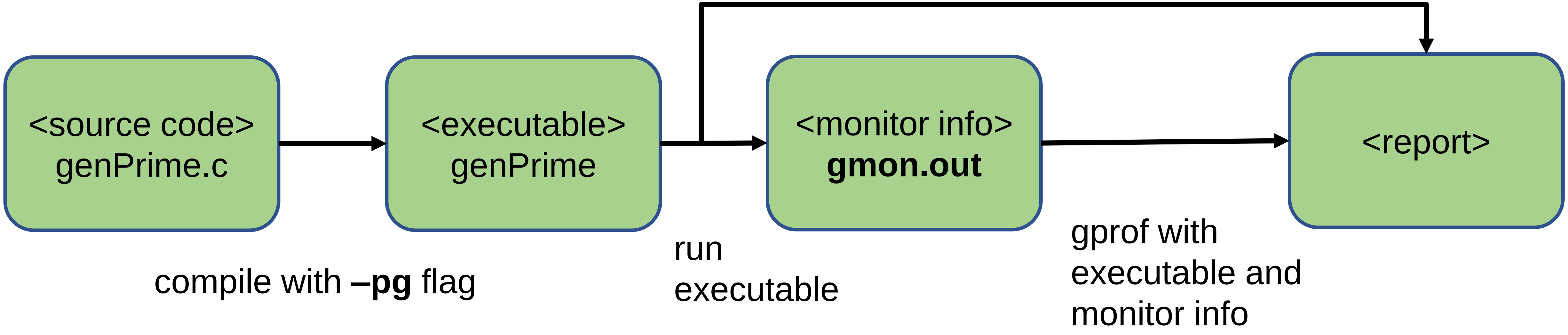
- **Loop Interchange**
  is used to improve cache performance and access patterns for multidimensional array in nested loops



GCC Command	Unoptimized	-O1	-O2	-O3
`$ gcc -o original matrixVector.c`	2.01	2.05	2.07	2.08
`$ gcc -o loop-interch matrixVector1.c`	0.27	0.08	0.06	0.06

Profiling Process with gprof

- $ gcc **–pg** genPrime.c –o genPrime –lm
- $ genPrime 10000000
- $ gprof genPrime **gmon.out**



Report by gpof (1)

the percentage of the total running time of the program used by this function

the cumulative running sum of the number of seconds accounted for this function and those listed above

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 85.48      2.92     2.92 10000017     0.00     0.00  isPrime
 13.49      3.38     0.46                             _init
  0.73      3.40     0.03   665025     0.00     0.00  getNextPrime
  0.29      3.41     0.01                             printArray
  0.00      3.41     0.00        1     0.00     0.00  allocateArray
  0.00      3.41     0.00        1     0.00     2.94  genPrimeSequence
  0.00      3.41     0.00        1     0.00     0.00  getTime
```

- flat profile
  - Sorted by the time spent

Report by gpof (2)

- call graph
  - parent
  - self
  - children

```
index % time    self  children    called     name
                0.03    2.92  665025/665025      genPrimeSequence [2]
[1]      86.2   0.03    2.92  665025         getNextPrime [1]
                2.92    0.00 10000017/10000017     isPrime [4]
-----------------------------------------------
                0.00    2.94       1/1           main [3]
[2]      86.2   0.00    2.94       1         genPrimeSequence [2]
                0.03    2.92  665025/665025      getNextPrime [1]
-----------------------------------------------
                                                 <spontaneous>
[3]      86.2   0.00    2.94                 main [3]
                0.00    2.94       1/1           genPrimeSequence [2]
                0.00    0.00       1/1           allocateArray [7]
                0.00    0.00       1/1           getTime [8]
-----------------------------------------------
                2.92    0.00 10000017/10000017     getNextPrime [1]
[4]      85.5   2.92    0.00 10000017         isPrime [4]
-----------------------------------------------
                                                 <spontaneous>
[5]      13.5   0.46    0.00                 _init [5]
-----------------------------------------------
                                                 <spontaneous>
[6]       0.3   0.01    0.00                 printArray [6]
-----------------------------------------------
                0.00    0.00       1/1           main [3]
[7]       0.0   0.00    0.00       1         allocateArray [7]
-----------------------------------------------
```

References

Dive into Systems

12. Code Optimization (https://diveintosystems.org/book/C12-CodeOpt/index.html)

GCC Optimization Flags (https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html)

Sample Codes (SystemProgramming/Dive-into-Systems-ch12)