Optymalizacja Kodu Na Różne Architektury
Ćwiczenie 8

1 Wprowadzenie

Celem laboratorium jest praktyczne zapoznanie się z technikami optymalizacji wykorzystania pamięci cache, jakimi są **tiling** (blokowanie pętli), **prefetching**, kopiowanie danych do lokalnych buforów oraz dopasowanie rozmiarów bloków danych do poziomów cache (L1, L2).

Algorytm mnożenia macierzy znajduje się w pliku:

Listing 1: Bazowa wersja kodu

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


static double gtod_ref_time_sec = 0.0;

/* Adapted from the bl2_clock() routine in the BLIS library */

double dclock()
{
  double the_time, norm_sec;
  struct timeval tv;
  gettimeofday( &tv, NULL );
  if ( gtod_ref_time_sec == 0.0 )
    gtod_ref_time_sec = ( double ) tv.tv_sec;
  norm_sec = ( double ) tv.tv_sec - gtod_ref_time_sec;
  the_time = norm_sec + tv.tv_usec * 1.0e-6;
  return the_time;
```

```
}


//subroutine for optimization
int mm(double ** first, double ** second, double ** multiply, int SIZE)
{
  int i,j,k;
  double sum = 0;
  for (i = 0; i < SIZE; i++) { //rows in multiply
    for (j = 0; j < SIZE; j++) { //columns in multiply
      for (k = 0; k < SIZE; k++) { //columns in first and rows in second
        sum = sum + first[i][k]*second[k][j];
      }
      multiply[i][j] += sum;
      sum = 0;
    }
  }
  return 0;
}



int main( int argc, const char* argv[] ){
  int i,j,iret;
  double ** first;
  double ** second;
  double ** multiply;

  double * first_;
  double * second_;
  double * multiply_;

  int optimal = 1;

  double dtime;

  if (argc == 3)
    optimal = atoi(argv[2]);;

  int SIZE = atoi(argv[1]);

//allocate blocks of continous memory
  first_ = (double*) malloc(SIZE*SIZE*sizeof(double));
```

```
  second_ = (double*) malloc(SIZE*SIZE*sizeof(double));
  multiply_ = (double*) malloc(SIZE*SIZE*sizeof(double));

//allocate 2D matrices
  first = (double**) malloc(SIZE*sizeof(double*));
  second = (double**) malloc(SIZE*sizeof(double*));
  multiply = (double**) malloc(SIZE*sizeof(double*));

  if(optimal){
//set pointers to continous blocks
    for (i = 0; i < SIZE; i++) {
      first[i] = first_ + i*SIZE;
      second[i] = second_ + i*SIZE;
      multiply[i] = multiply_ + i*SIZE;
    }
  }
  else{
    for (i = 0; i < SIZE; i++) {
      first[i] = (double*) malloc(SIZE*sizeof(double));
      second[i] = (double*) malloc(SIZE*sizeof(double));
      multiply[i] = (double*) malloc(SIZE*sizeof(double));
    }
  }


//fill matrices with test data
  for (i = 0; i < SIZE; i++) { //rows in first
    for (j = 0; j < SIZE; j++) { //columns in first
      first[i][j]=i+j;
      second[i][j]=i-j;
      multiply[i][j]=0;
    }
  }

//measure mm subroutine computation time
  dtime = dclock();
  iret = mm(first,second,multiply,SIZE);
  dtime = dclock()-dtime;
  printf( "Time: %le \n", dtime);


  fflush( stdout );
```

```
//cleanup
  free(first_);
  free(second_);
  free(multiply_);

  if(!optimal){
    for (i = 0; i < SIZE; i++) {
      free(first[i]);
      free(second[i]);
      free(multiply[i]);
    }
  }

  free(first);
  free(second);
  free(multiply);

  return iret;
}
```

2 Ćwiczenia

Przy kolejnych krokach proszę wykonywać kopie aktualnych wersji stanu kodu. Będą potrzebne przy ostatnim punkcie oraz na kolejnych zajęciach. Proszę na bieżąco notować czasy wykonywania się obliczeń.

2.1

```
maciekw@Banach:~lab8> gcc mm1.c -O2
maciekw@Banach:~lab8> ./a.out 2000
Time: 9.418615e+00
maciekw@Banach:~/studenci/OORA/skrypty/lab8>
```

2.2

Zmodyfikuj kod mnożenia macierzy, dodając blokowanie pętli (tiling). Przykładowy fragment kodu:

```
#define BLOCK_SIZE 32

  for (ii = 0; ii < SIZE; ii += BLOCK_SIZE) {
    for (jj = 0; jj < SIZE; jj += BLOCK_SIZE) {
      for (kk = 0; kk < SIZE; kk += BLOCK_SIZE) {
        for (i = ii; i < ii+BLOCK_SIZE && i<SIZE; i++) {
          for (j = jj; j < jj+BLOCK_SIZE && j<SIZE; j++) {
            for (k = kk; k < kk+BLOCK_SIZE && k<SIZE; k++) {
              sum = sum + first[i][k]*second[k][j];
            }
            multiply[i][j] += sum;
            sum = 0;
          }
        }
      }
    }
  }
```

Dobierz rozmiar bloku (BLOCK_SIZE) odpowiedni dla cache L2 (np. 32, 64, 128). Przeprowadź pomiar czasu wykonania i zanotuj wyniki.

2.3

Dodaj prefetching do powyższej implementacji:

```
#include <xmmintrin.h>

#define PREFETCH_DISTANCE 64

for (j = jj; j < jj+BLOCK_SIZE && j<SIZE; j++) {
    if (i + PREFETCH_DISTANCE < SIZE)
        _mm_prefetch((double*)&first[i + PREFETCH_DISTANCE][kk],
          _MM_HINT_T0);
    for (k = kk; k < kk+BLOCK_SIZE && k<SIZE; k++) {
```

Zintegruj prefetching z implementacją blokową. Przeprowadź testy czasowe, porównaj wyniki.

2.4

Zaimplementuj lokalną reorganizację macierzy, linearyzując dostęp do pamięci. Usuń polecenie **_mm_prefetch**.

```
double local_first[BLOCK_SIZE * BLOCK_SIZE];

for (kk = 0; kk < SIZE; kk += BLOCK_SIZE) {
  for (i = 0; i < BLOCK_SIZE && (ii + i) < SIZE; i++) {
    for (j = 0; j < BLOCK_SIZE && (kk + j) < SIZE; j++) {
      local_first[i * BLOCK_SIZE + j] = first[ii + i][kk + j];
    }
  }

sum += local_first[(i - ii) * BLOCK_SIZE + (k - kk)] * second[k][j];
```

Przeprowadź pomiary wydajności.

2.5

Wykonaj analogiczną modyfikację dla macierzy **second**. Przeprowadź pomiary wydajności.

2.6

Skoryguj liniowość przechodzenia po macierzach.

```
for (i = ii; i < ii + BLOCK_SIZE && i < SIZE; i++) {
  for (k = kk; k < kk + BLOCK_SIZE && k < SIZE; k++) {
    a_val = local_first[(i - ii) * BLOCK_SIZE + (k - kk)];
      for (j = jj; j < jj + BLOCK_SIZE && j < SIZE; j++) {
        multiply[i][j] += a_val * local_second[(k - kk)
          * BLOCK_SIZE + (j - jj)];
      }
    }
  }
```

Przeprowadź pomiary wydajności.

2.7

Przeorganizuj pętle tak, aby macierz **first** była kopiowana tylko raz.

```
for (kk = 0; kk < SIZE; kk += BLOCK_SIZE) {
  for (ii = 0; ii < SIZE; ii += BLOCK_SIZE) {
    for (i = 0; i < BLOCK_SIZE && (ii + i) < SIZE; i++) {
      for (j = 0; j < BLOCK_SIZE && (kk + j) < SIZE; j++) {
        local_first[i * BLOCK_SIZE + j] = first[ii + i][kk + j];
      }
    }
    for (jj = 0; jj < SIZE; jj += BLOCK_SIZE) {
```

Przeprowadź pomiary wydajności.

2.8

Wychodząc od wersji 6, dodaj lokalną linearną macierz **multiply**.

```
for (jj = 0; jj < SIZE; jj += BLOCK_SIZE) {
    for (i = 0; i < BLOCK_SIZE && (ii + i) < SIZE; i++) {
      for (j = 0; j < BLOCK_SIZE && (jj + j) < SIZE; j++) {
        local_multiply[i * BLOCK_SIZE + j] = 0;
      }
    }
  for (kk = 0; kk < SIZE; kk += BLOCK_SIZE) {

....

for (i = 0; i < BLOCK_SIZE && (ii + i) < SIZE; i++) {
  for (k = 0; k < BLOCK_SIZE && (kk + k) < SIZE; k++) {
    a_val = local_first[i * BLOCK_SIZE + k];
    for (j = 0; j < BLOCK_SIZE && (jj + j) < SIZE; j++) {
      local_multiply[i * BLOCK_SIZE + j] += a_val *
          local_second[k * BLOCK_SIZE + j];
    }

...

for (i = 0; i < BLOCK_SIZE && (ii + i) < SIZE; i++) {
  for (j = 0; j < BLOCK_SIZE && (jj + j) < SIZE; j++) {
    multiply[ii + i][jj + j] = local_multiply[i * BLOCK_SIZE + j];
  }
}
```

Przeprowadź pomiary wydajności.

3 Podsumowanie

Które metody optymalizacji dały najlepsze rezultaty?
Dlaczego? Jakie mechanizmy za nimi stały?