Computação de Alto Desempenho – COC472
Trabalho 03 – Roofline

Lucas Tavares Da Silva Ferreira | DRE 120152739

Engenharia de Computação e Informação - Universidade Federal do Rio de Janeiro (UFRJ)
Rio de Janeiro – RJ – Brasil

GitHub: https://github.com/lucastavarex/hpc

1. Introdução

Em resumo, o Modelo de Teto (Roofline Model) é uma representação gráfica que ajuda a entender e visualizar o desempenho máximo alcançável por um programa em relação aos recursos de hardware disponíveis. Ele é usado principalmente para avaliar e otimizar o desempenho de algoritmos e identificar possíveis gargalos em um sistema. Neste trabalho, a atividade proposta será gerar o roofline do loop onde está a maior parte do trabalho do código da última tarefa.

As especificações sobre o processador da máquina utilizada são:

Intel(R) Core(TM) i3 CPU M 370 @ 2.40GHz
Logical CPU Count 4
Sistema operacional Linux

2. Tetos de desempenho

Utilizando a ferramenta Intel Advisor, foi possível plotar o gráfico de tetos de desempenho a seguir:

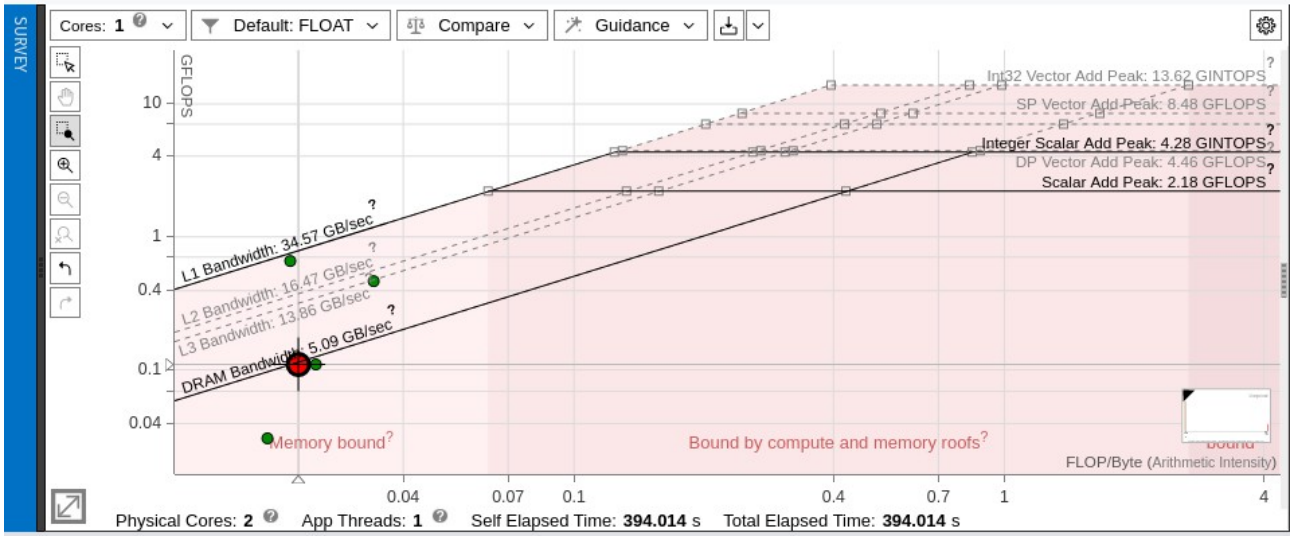

Figura 1: Roofline (Código sem otimização (n=2000))

3. Desempenho do código

Como pode-se observar na aba de resumo do Intel Advisor, o FP Arithmetic Intensity é AI = 0.023

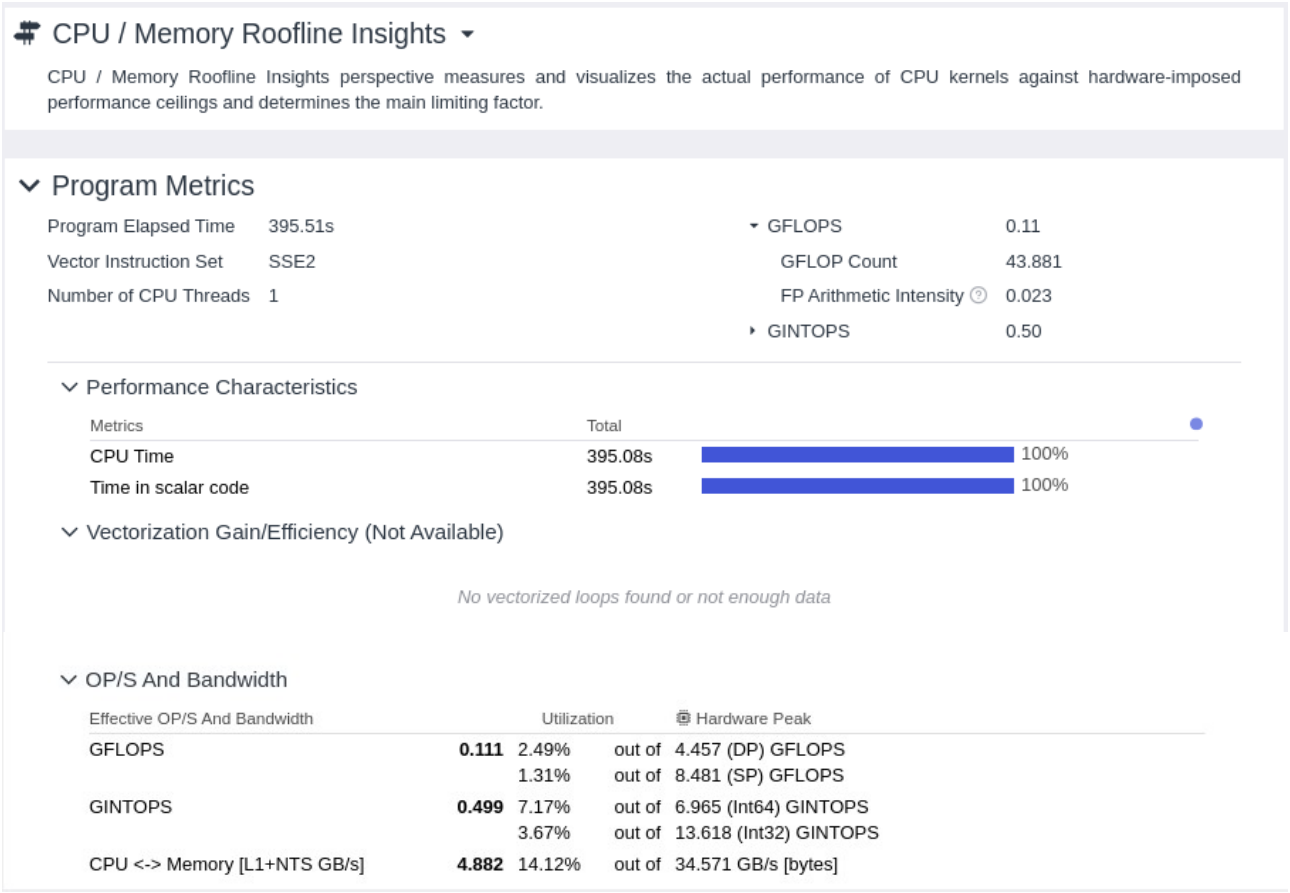

Figura 3: Resumo de desempenho (Código sem otimização (n=2000))

4. Código utilizado

```
//
// Implementation of the iterative Jacobi method.
//
// Given a known, diagonally dominant matrix A and a known vector b, we aim
to
// to find the vector x that satisfies the following equation:
//
// Ax = b
//
// We first split the matrix A into the diagonal D and the remainder R:
//
// (D + R)x = b
//
// We then rearrange to form an iterative solution:
//
```

```
// x' = (b - Rx) / D
//
// More information:
// -> https://en.wikipedia.org/wiki/Jacobi_method
//

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

static int N;
static int MAX_ITERATIONS;
static int SEED;
static double CONVERGENCE_THRESHOLD;

#define SEPARATOR "------------------------------------\n"

// Return the current time in seconds since the Epoch
double get_timestamp();

// Parse command line arguments to set solver parameters
void parse_arguments(int argc, char *argv[]);

// Run the Jacobi solver
// Returns the number of iterations performed
int run(double *A, double *b, double *x, double *xtmp)
{
int itr;
int row, col;
double dot;
double diff;
double sqdiff;
double *ptrtmp;

// Loop until converged or maximum iterations reached
itr = 0;
do
{
// Perfom Jacobi iteration
for (row = 0; row < N; row++)
{
dot = 0.0;
for (col = 0; col < N; col++)
{
if (row != col)
dot += A[row + col*N] * x[col];
}
xtmp[row] = (b[row] - dot) / A[row + row*N];
}
```

```
// Swap pointers
ptrtmp = x;
x = xtmp;
xtmp = ptrtmp;

// Check for convergence
sqdiff = 0.0;
for (row = 0; row < N; row++)
{
diff = xtmp[row] - x[row];
sqdiff += diff * diff;
}

itr++;
} while ((itr < MAX_ITERATIONS) && (sqrt(sqdiff) > CONVERGENCE_THRESHOLD));

return itr;
}

int main(int argc, char *argv[])
{
parse_arguments(argc, argv);

double *A = malloc(N*N*sizeof(double));
double *b = malloc(N*sizeof(double));
double *x = malloc(N*sizeof(double));
double *xtmp = malloc(N*sizeof(double));

printf(SEPARATOR);
printf("Matrix size: %dx%d\n", N, N);
printf("Maximum iterations: %d\n", MAX_ITERATIONS);
printf("Convergence threshold: %lf\n", CONVERGENCE_THRESHOLD);
printf(SEPARATOR);

double total_start = get_timestamp();

// Initialize data
srand(SEED);
for (int row = 0; row < N; row++)
{
double rowsum = 0.0;
for (int col = 0; col < N; col++)
{
double value = rand()/(double)RAND_MAX;
A[row + col*N] = value;
rowsum += value;
}
A[row + row*N] += rowsum;
b[row] = rand()/(double)RAND_MAX;
x[row] = 0.0;
```

```
}

// Run Jacobi solver
double solve_start = get_timestamp();
int itr = run(A, b, x, xtmp);
double solve_end = get_timestamp();

// Check error of final solution
double err = 0.0;
for (int row = 0; row < N; row++)
{
double tmp = 0.0;
for (int col = 0; col < N; col++)
{
tmp += A[row + col*N] * x[col];
}
tmp = b[row] - tmp;
err += tmp*tmp;
}
err = sqrt(err);

double total_end = get_timestamp();

printf("Solution error = %lf\n", err);
printf("Iterations = %d\n", itr);
printf("Total runtime = %lf seconds\n", (total_end-total_start));
printf("Solver runtime = %lf seconds\n", (solve_end-solve_start));
if (itr == MAX_ITERATIONS)
printf("WARNING: solution did not converge\n");
printf(SEPARATOR);

free(A);
free(b);
free(x);
free(xtmp);

return 0;
}

double get_timestamp()
{
struct timeval tv;
gettimeofday(&tv, NULL);
return tv.tv_sec + tv.tv_usec*1e-6;
}

int parse_int(const char *str)
{
char *next;
int value = strtoul(str, &next, 10);
return strlen(next) ? -1 : value;
```

```
}

double parse_double(const char *str)
{
char *next;
double value = strtod(str, &next);
return strlen(next) ? -1 : value;
}

void parse_arguments(int argc, char *argv[])
{
// Set default values
N = 2000;
MAX_ITERATIONS = 20000;
CONVERGENCE_THRESHOLD = 0.0001;
SEED = 0;

for (int i = 1; i < argc; i++)
{
if (!strcmp(argv[i], "--convergence") || !strcmp(argv[i], "-c"))
{
if (++i >= argc || (CONVERGENCE_THRESHOLD = parse_double(argv[i])) < 0)
{
printf("Invalid convergence threshold\n");
exit(1);
}
}
else if (!strcmp(argv[i], "--iterations") || !strcmp(argv[i], "-i"))
{
if (++i >= argc || (MAX_ITERATIONS = parse_int(argv[i])) < 0)
{
printf("Invalid number of iterations\n");
exit(1);
}
}
else if (!strcmp(argv[i], "--norder") || !strcmp(argv[i], "-n"))
{
if (++i >= argc || (N = parse_int(argv[i])) < 0)
{
printf("Invalid matrix order\n");
exit(1);
}
}
else if (!strcmp(argv[i], "--seed") || !strcmp(argv[i], "-s"))
{
if (++i >= argc || (SEED = parse_int(argv[i])) < 0)
{
printf("Invalid seed\n");
exit(1);
}
}
```

```
else if (!strcmp(argv[i], "--help") || !strcmp(argv[i], "-h"))
{
printf("\n");
printf("Usage: ./jacobi [OPTIONS]\n\n");
printf("Options:\n");
printf(" -h --help Print this message\n");
printf(" -c --convergence C Set convergence threshold\n");
printf(" -i --iterations I Set maximum number of iterations\n");
printf(" -n --norder N Set maxtrix order\n");
printf(" -s --seed S Set random number seed\n");
printf("\n");
exit(0);
}
else
{
printf("Unrecognized argument '%s' (try '--help')\n", argv[i]);
exit(1);
}
}
}
```