

گزارش پروژه برنامه‌نویسی چندهسته‌ای

دکتر ممتازپور

حسین زارع‌دار ۹۶۳۱۰۳۵

نیمسال دوم ۹۸-۹۹

بستر اجرای پروژه

CPU: Intel Corei7 7700HQ (4 physical cores, 8 logical cores)

GPU: NVIDIA GeForce GTX 1050 (Notebook) (5 SMs, 4 GB of Memory)

Main Memory: 16GB

OS: Ubuntu 18.04

ورودی و خروجی برنامه

ورودی برنامه تعداد فایل است که در پوشه‌ی `data_in` در کنار فایل اجرایی قرار می‌گیرد. در هر فایل ۱ یا بیشتر از ۱ ماتریس به صورت `row major` ذخیره شده است که ماتریس‌ها با '\n' از یکدیگر جدا شده است.

خروجی برنامه، به ازای هر فایل ورودی، یک فایل در پوشه‌ی `data_out` ایجاد می‌شود که حاصل دترمینان ماتریس‌های فایل ورودی، در آن نوشته شده است.

نمونه‌ی کوچکی از ورودی و خروجی در کنار پروژه قرار دارد. ورودی دو فایل است که در هر فایل، ۳ ماتریس ۴ در ۴ قرار گرفته است.

قدم اول: کد سریال

در قدم اول، کد سریال محاسبه‌ی دترمینان پیاده‌سازی شد. نتیجه پروفایلینگ `Hotspots` به ازای ورودی ۲۰ فایل، هر کدام دارای ۱ ماتریس ۵۱۲ در ۵۱۲، به صورت زیر می‌باشد:

Hotspots

Hotspots by CPU Utilization

?

INTEL VTUNE AMPLIFIER 2019

Analysis Configuration

Collection Log

Summary

Bottom-up

Caller/Callee

Top-down Tree

Platform

serial_determinant.c

isoc99_sscanf.c

Grouping: Function / Call Stack

Function / Call Stack	CPU Time					Spin Time	Overhead Time	Module	Function (Full)	Source File	
	Effective Time by Utilization										
	Idle	Poor	Ok	Ideal	Over						
determinant	150.000ms					0ms	0ms	a.out	determinant	serial_determinant.c	
__isoc99_sscanf	67.986ms					0ms	0ms	libc.so.6	__isoc99_sscanf	isoc99_sscanf.c	
_IO_fgets	12.014ms						0ms	0ms	libc.so.6	_IO_fgets	iofgets.c

همانطور که مشاهده می‌شود، تقریباً یک‌سوم زمان اجرا به عملیات خواندن از فایل و ذخیره در حافظه، و بقیه زمان به محاسبات اختصاص دارد.

در قسمت محاسبات داریم:

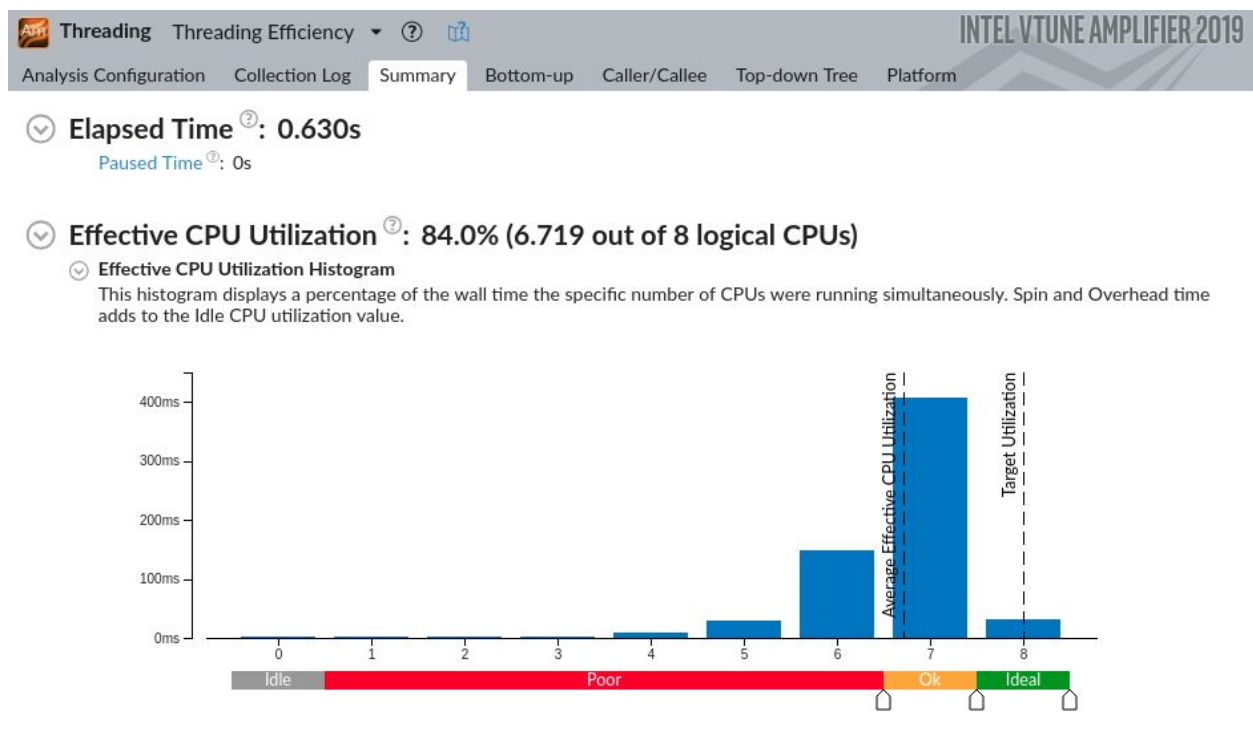
Hotspots Hotspots by CPU Utilization							INTEL VTUNE AMPLIFIER 2019	
Analysis Configuration Collection Log Summary Bottom-up Caller/Callee Top-down Tree Platform serial_determinant.c x isoc99_sscanf.c x								
Source Assembly								
S...	Source					CPU Time: Total	CPU Time: Self	
28	// looping through diagonal entries							
29	for (int i = 0; i < n; i++) {							
30								
31	// finding a non-zero entry below current diagonal entry							
32	int k = i;							
33	while (matrix[k * n + i] == 0 && k < n)							
34	k++;							
35								
36	// if nothing was found, then determinant is 0							
37	if (matrix[k * n + i] == 0)							
38	return 0;							
39								
40	// if k != i, then swap the rows							
41	if (k != i) {							
42	swap *= -1;							
43	for (int j = 0; j < n; j++) {							
44	double temp = matrix[i * n + j];							
45	matrix[i * n + j] = matrix[k * n + j];							
46	matrix[k * n + j] = temp;							
47	}							
48	}							
49								
50	det *= matrix[i * n + i];							
51								
52	// making the entries below diagonal entry 0, using row operations							
53	for (int j = i + 1; j < n; j++) {							
54								
55	double factor = -1 * matrix[j * n + i] / matrix[i * n + i];							
56	matrix[j * n + i] = 0;							
57								
58	for (int k = i + 1; k < n; k++) {					4.3%	10.002ms	
59	matrix[j * n + k] += factor * matrix[i * n + k];					60.9%	139.998ms	
60	}							
61	}							

در محاسبه‌ی دترمینان به کمک تجزیه‌ی LU، حلقه‌ای که وظیفه‌ی انجام عملیات سطری را بر عهده دارد، بیشترین زمان اجرا را به خود اختصاص داده است.

قدم دوم: موازی سازی پردازش فایل ها به کمک OpenMP

در این قدم به کمک OpenMP، یک نخ به عنوان master، فایل های درون پوشه ی data_in را شناسایی می کند، سپس به ازای هر فایل یک task ایجاد کرده که نخ ها آن ها را انجام می دهند. هر نخ تسکی را برمی دارد و وظیفه ی خواندن یک فایل، محاسبه ی دترمینان ماتریس های درون آن فایل و نوشتن نتایجش را بر عهده می گیرد.

تحلیل Threading ازای ورودی ۲۰ فایل، هر کدام دارای ۱ ماتریس ۵۱۲ در ۵۱۲، به صورت زیر می باشد:



Threading Efficiency - Bottom-up						
Grouping: Thread / Function / Call Stack						
Thread / Function / Call Stack	CPU Time			Wait Time by Utilization		Wait Count
	Effective Time by Utilization	Spin Time	Overhead Time	Idle	Poor	
func@0x16840 (TID: 20367)	590.000ms	0ms	0ms	0.164ms		4
func@0x16840 (TID: 20369)	590.000ms	0ms	0ms	0.270ms		4
func@0x16840 (TID: 20372)	580.000ms	0ms	0ms	0.090ms		3
func@0x16840 (TID: 20368)	570.000ms	0ms	0ms	0.060ms		2
a.out (TID: 20351)	560.000ms	0ms	0ms	0.122ms		3
func@0x16840 (TID: 20370)	490.000ms	0ms	0ms	0.108ms		3
func@0x16840 (TID: 20373)	490.000ms	0ms	0ms	0.133ms		4
func@0x16840 (TID: 20371)	360.000ms	0ms	0ms	12.973ms		6

که مشاهده می‌شود، از هسته‌های پردازنده به شکل مناسبی استفاده شده است.

قدم سوم: محاسبه دترمینان به کمک پردازنده گرافیکی

در این قدم علاوه بر اقدامات مرحله قبل، یعنی از استفاده از OpenMP Task برای سپردن پردازش هر فایل به یک نخ پردازنده، هر نخ برای محاسبه دترمینان ماتریس‌های مربوط به فایل خود، از پردازنده گرافیکی بهره می‌برد. برای محاسبه دترمینان در GPU، از همان تجزیه LU استفاده شده است، و برای مراحل مختلف آن، از توابع کتابخانه‌ی cuBLAS استفاده شده است.

مقایسه و تحلیل زمان اجرا

زمان اجرا ۳ نسخه‌ی برنامه (در حالت Release) از ابتدا تا انتهای آن (یعنی خواندن تمامی فایل‌ها، انجام محاسبات و نوشتن نتایج در فایل)، به ازای ورودی‌های مختلف به صورت زیر می‌باشد:

ورودی	Serial time	OMP Task		OMP Task + CUDA	
		time	speedup	time	speedup
۲۰۰ فایل، در هر فایل یک ماتریس ۲۵۶ در ۲۵۶	4.84s	1.43s	3.13	3.22s	1.50
۷۰ فایل، در هر فایل یک ماتریس ۵۱۲ در ۵۱۲	6.95s	2.60s	2.67	3.81	1.82
۵۰ فایل، در هر فایل یک ماتریس ۱۰۲۴ در ۱۰۲۴	23.14s	12.64s	1.83	10.23s	2.26
۲۰ فایل، در هر فایل یک ماتریس ۲۰۴۸ در ۲۰۴۸	59.6s	43.12s	1.28	21.45	2.77

همانطور که مشاهده می‌شود، در اندازه‌های کوچک ماتریس، استفاده از موازی‌سازی در CPU به کمک OMP Task، تسریع مناسبی را ایجاد می‌کند اما استفاده از GPU برای محاسبه‌ی دترمینان ماتریس‌ها، به علت سربار جابه‌جایی داده، تسریع را کاهش می‌دهد. ما با افزایش سایز ماتریس‌ها، مشاهده می‌کنیم که استفاده از GPU و CPU به طور همزمان، تسریع را به شکل خوبی افزایش می‌دهد.