به نام خدا

گزارش پروژه‌ی نـهایی بـرنـامـه‌نـویـسی چندهسـته‌ای

امیرحسین بینش ۹۵۳۱۰۱۲

ویژگی‌های سیستم استفاده شده

Processor: Intel Core i7 7700 HQ 2.80 GHz - 4 cores and threads

Memory: 16 GB DDR4

Graphics Card: NVIDIA GeForce GTX 1050 - 4 GB GDDR5

کد سریال

راه اول: دترمینان بازگشتی (materials/Recursive.cpp)

همانطور که در صورت پروژه گفته شد، استفاده از روش بازگشتی برای محاسبه‌ی دترمینان ماتریس‌های بزرگتر از ۱۰ در ۱۰ عملا غیر ممکن است. باتوجه به اینکه در این مسئله ما با تعداد زیادی ماتریس و در سایزهای متفاوت تا ۶۴ در ۶۴ کار داریم، با روش بازگشتی نمی‌توان به سرعت به نتیجه رسید.

```
int determinant(DataSet dataSet, int* indices, int size) {
    int i;
    int result = 0;
    if (size == 2) {
        result = dataSet.A[indices[0]] * dataSet.A[indices[1] + dataSet.n] -
                 dataSet.A[indices[1]] * dataSet.A[indices[0] + dataSet.n];
        return result;
    }
    else {
        for (i = 0; i < size; i++) {
            int* newIndices = make(indices, i, size, dataSet.n);
            if (i % 2 == 0) {
                result += dataSet.A[indices[i]] *
                          determinant(dataSet, newIndices, size - 1);
            }
            else {
                result -= dataSet.A[indices[i]] *
                          determinant(dataSet, newIndices, size - 1);
            }
            free(newIndices);
        }
    }
    return result;
}پ
```

کد بالا برای ماتریس ۱۶ در ۱۶ بعد از بیش از ۴۰ دقیقه جواب نداد؛ چون پیچیدگی O(n!) دارد.

راه دوم: تجزیه LU

در روش تجزیه‌ی LU نیازی به محاسبه‌ی بازگشتی نیست و با پیچیدگی در بدترین حالت $O(n^3)$ می‌توان یک ماتریس n در n را تجزیه کرد. برای محاسبه‌ی دترمینان ماتریس A، از قواعد زیر استفاده می‌کنیم:

$$A = L * U$$

$$\det(A) = det(L) * \det(U)$$

$$\det(L) = \prod L_{i,i} = 1, \quad \det(U) = \prod U_{i,i}$$

$$\rightarrow \det(A) = \prod U_{i,i}$$

پس فقط کافی‌است U را پیدا کنیم که ماتریس بالامثلثی است که از عملیات elimination ماتریس A بدست می‌آید. در اینجا از روش حذف گاوسی استفاده می‌شود.

پیاده‌سازی یک (materials/LU Sort.cpp)

```
float determinantLU(DataSet* dataset) {
    int i, j;       // Iterators
    float det = 1;  // Determinant to be calculated

    // Do for each column
    for (i = 0; i < dataset->getN() - 1; i++) {
        det *= dataset->getU(i, i);
        columnElimination(dataset, i);
    }

    // Multiply result by last pivot element
    det *= dataset->getU(dataset->getN() - 1, dataset->getN() - 1);
    return det;
}
```

با این روش برای دو فایل شامل ۲۰ ماتریس، که ده تای آنها ۳۲ در ۳۲ و ده تای دیگر ۶۴ در ۶۴ هستند به زمان میانگین ۲۳۱ میلی‌ثانیه می‌رسیم. نتایج چاپ شده برای فایل اول به شکل زیر هستند:

```
-1744458680609484681627537139302400.000000

2693385756616939587416688668180480.000000

-nan(ind)

-nan(ind)

-nan(ind)

-2591934539402125721311691931648O.000000

-4024296253421438751004744839331840.000000

1191238993088503226954389703884800.000000
```

```
-327907917429111414971429050908672.000000
861873391591726719792233536552 96.000000
```

همانطور که دیده شد، این روش گاهی اوقات جواب درست نمی‌دهد. با تست برای ماتریس‌های کوچکتر چک شد که جواب‌های بدست آمده درست هستند.

دلیل جواب‌های غلط الگوریتم بالا، این است که اگر عناصر محوری ماتریس را فقط درایه‌های قطری بگیریم، ممکن است به اتفاق صفر باشند/شده باشند و این باعث می‌شود در محاسبه‌ی ضریب به تقسیم صفر بخوریم و ماتریس U تا آخر غلط شود.

برای حل این مشکل باید تجزیه LU را با روش partial pivoting حساب کنیم. این روش بصورت زیر است:

$$P * A = L * U \; where \; P \; is \; permutation \; matrix$$

$$\det(P) * \det(A) = \prod U_{i,i}$$

$$\det(P) = (swap \; counts \; mod \; 2) * \; -1$$

$$\rightarrow \det(A) = \; (swap \; counts \; mod \; 2) * (-1) * \prod U_{i,i}$$

<u>پیاده‌سازی دو</u>

در این روش، برای هر ستون که در حال تجزیه‌ی آن هستیم، بر اساس عناصر آن ستون، سطرها را از بزرگ به کوچک مرتب می‌کنیم (مرتب سازی براساس مقدار مطلق و بدون دستکاری سطرهای بالایی) و سپس برای آن ستون عملیات حذف گاوسی را انجام می‌دهیم.

```
float determinantLU(DataSet* dataset) {
    int i, j;       // Iterators
    float det = 1;  // Determinant to be calculated

    // Do for each column
    for (i = 0; i < dataset->getN() - 1; i++) {
        // Sort and swap rows below
        // Negative the result times swapping happened (sortnswap returns it)
        det *= sortnswap(i, dataset);
        // Multiply result by the pivot element
        det *= dataset->getU(i, i);
        // Do the Guassian Elimination for this column
        columnElimination(dataset, i);
    }

    // Multiply result by last pivot element
    det *= dataset->getU(dataset->getN() - 1, dataset->getN() - 1);
    return det;
}
```

با این روش همه‌ی جواب‌ها درست بودند ولی سرعت به شدت کاهش یافت. میانگین سرعت برای همان داده-ها به ۱۲۶۵ میلی‌ثانیه رسید.

برای بهبود سرعت راهکارهای زیر به ترتیب پیاده‌سازی شدند:

- بهبود یک: فقط وقتی swapnsort را صدا می‌زنیم که عنصر محوری صفر باشد.
- بهبود دو: بجای swapnsort تابع جدید swapMax را می‌سازیم که بجای مرتب سازی، بزرگترین مقدار مطلق آن ستون را پیدا می کند و جایگزین می‌کند.
- بهبود سه: بجای swap کردن سطرها بصورت واقعی، یک آرایه مجازی از اندیس‌ها نگهداری می‌کنیم و فقط آن را عوض می‌کنیم.(Source Codes/SERIAL)

```
float determinantLU(DataSet* dataset) {
    int i, j;       // Iterators
    float det = 1;  // Determinant to be calculated
    int* indices;   // Virtual holder of row positions

    // Load Default indices : [0, 1, ... , n]
    indices = getDefaultIndices(dataset->getN());

    // Do for each column
    for (i = 0; i < dataset->getN() - 1; i++) {
        // Find max absolute number of this column and call it pivot
        // Negative the result if a swapping happened
        det *= swapMax(i, dataset, indices);
        // Multiply result by the pivot element
        det *= dataset->getU(indices[i], i);
        // Do the Guassian Elimination for this column
        columnElimination(dataset, i, indices);
    }

    // Multiply result by last pivot element
    det *= dataset->getU(indices[dataset->getN() - 1], dataset->getN() - 1);
    return det;
}
```

با این سه راهکار، به زمان میانگین ۲۳۸ میلی‌ثانیه می‌رسیم که بسیار نزدیک به پیاده‌سازی اول است.

حالا برای مقایسه‌ی روش‌های بالا بر روی ۳۲ فایل تست کیس الگوریتم‌ها را اجرا می‌کنیم.

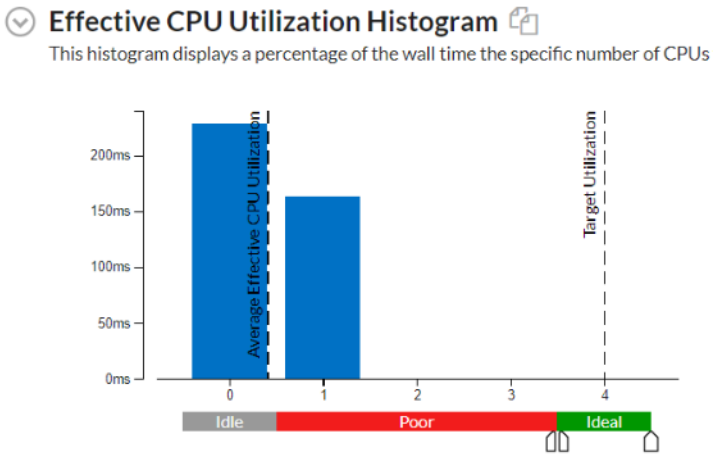
جدول مقایسه زمان اجرای روش‌های پیاده سازی شده:

پیاده سازی	پیاده‌سازی یک	پیاده‌سازی دو	بهبود یک	بهبود دو	بهبود سه
زمان(ms)Debug	1520	3233	2450	1706	1596
زمان(ms)Release	141	-	-	-	153
صحت جواب	خیر	بلی	بلی	بلی	بلی

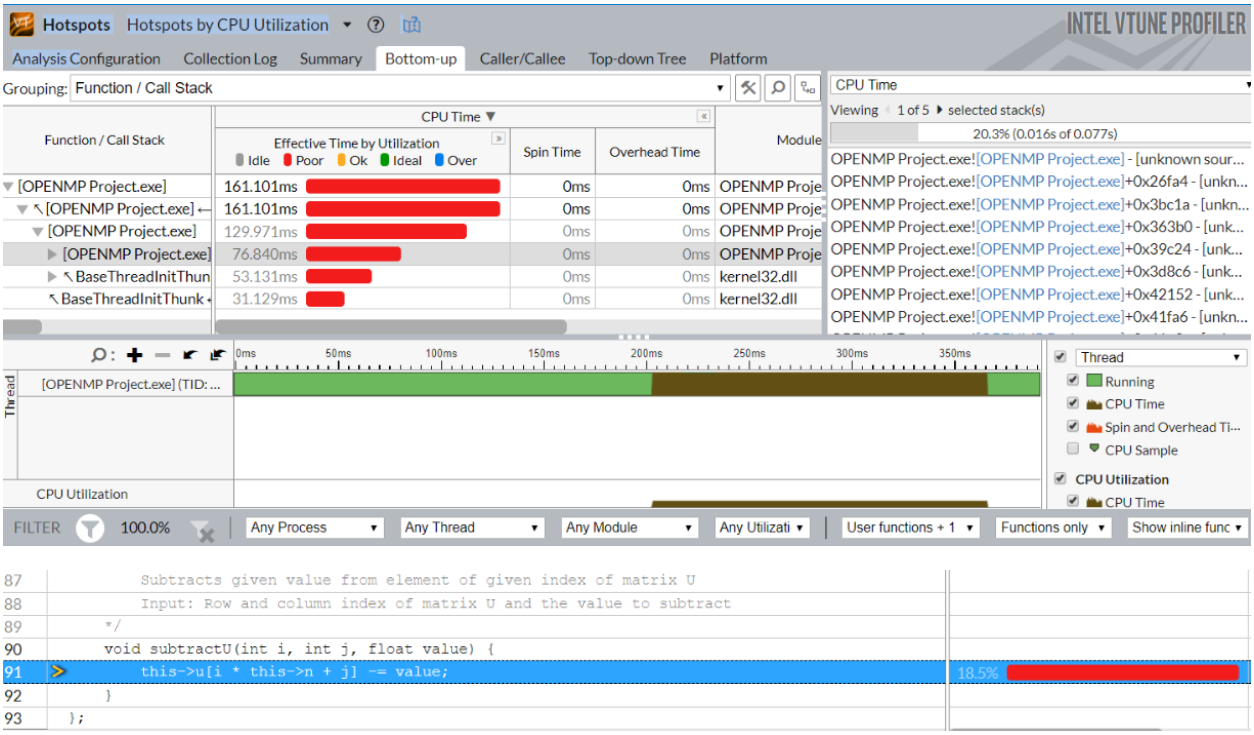
* تست های زمان از این به بعد، بر روی ۳۲ فایل تست کیس با مجموع حدود ۷۰۰ ماتریس با سایزهای مختلف انجام شده اند و پروفایل ها روی دو فایل با ۲۰ ماتریس ۳۲ در ۳۲ و ۶۴ در ۶۴ انجام شده اند.

پروفایل کد سریال

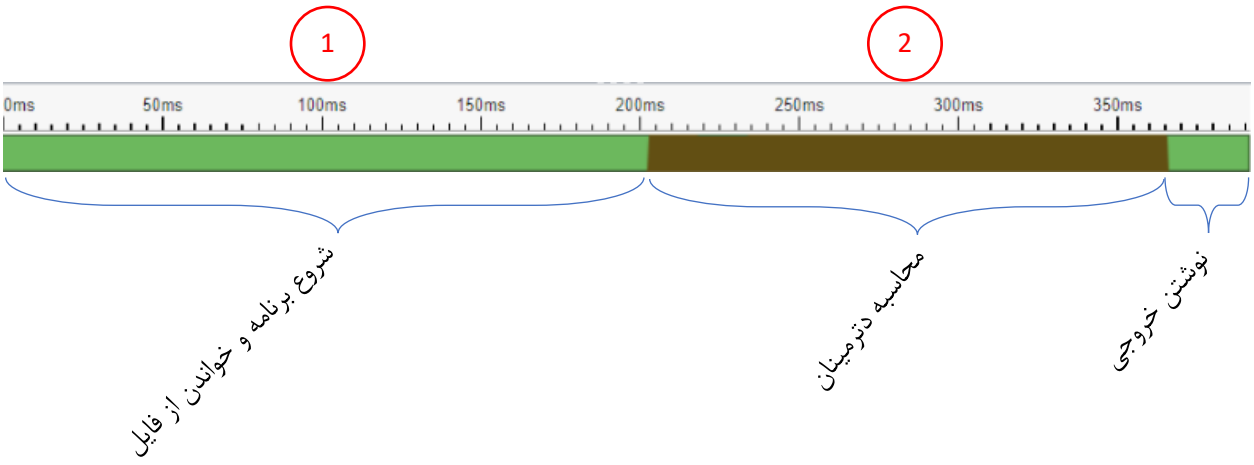
در این مرحله بهترین کد سریال (بهبود ۳) را پروفایل می‌کنیم.



همانطور که از تصویر پیداست، بیش از نصف زمان، صرف خواندن از فایل و تبدیل متن به ماتریس می‌شود که عملا سی‌پی‌یو در این مدت بیکار است.



در اینجا نیز مشاهده می‌شود در این پروفایل(با ماتریس‌های کوچک‌تر و تعداد فایل‌های بیشتر) فقط ۱۸ درصد کار سی‌پی‌یو عملیات کاهش سطری بوده.



گلوگاه اصلی: خواندن فایل‌ها و تبدیل به ماتریس کند است.

گلوگاه بعدی: یک هسته زمان زیادی می‌برد تا دترمینان را حساب کند؛ خصوصا اگر بزرگ باشد.

با توجه به نمودار بالا، برای سرعت بخشیدن، راهکار زیر استفاده می‌شود.

1. چند نخ، فایل‌ها را می‌خوانند.
2. دترمینان را کارت گرافیکی حساب می‌کند.

روش‌های موازی سازی:

- **روش اول(روش اصلی پیاده‌سازی شده)**: هر نخ یک ماتریس را حساب می‌کند.
- روش دوم: هر نخ یک فایل را می‌خواند، حساب می‌کند و جواب را می‌نویسد.
- روش سوم: یک نخ فایل‌ها را می‌خواند و با الگوی تولیدکننده و مصرف‌کننده دترمینان‌ها را حل می‌کنیم.
- روش چهارم: چند نخ فایل‌ها را می‌خوانند و کارت گرافیکی آن‌ها را حل می‌کنند.
- روش پنجم: استفاده از چند روش بصورت هایبرید.

کد موازی

روش اول: هر نخ یک ماتریس را حساب می‌کند.(`Source Codes/PARALLEL 1`)

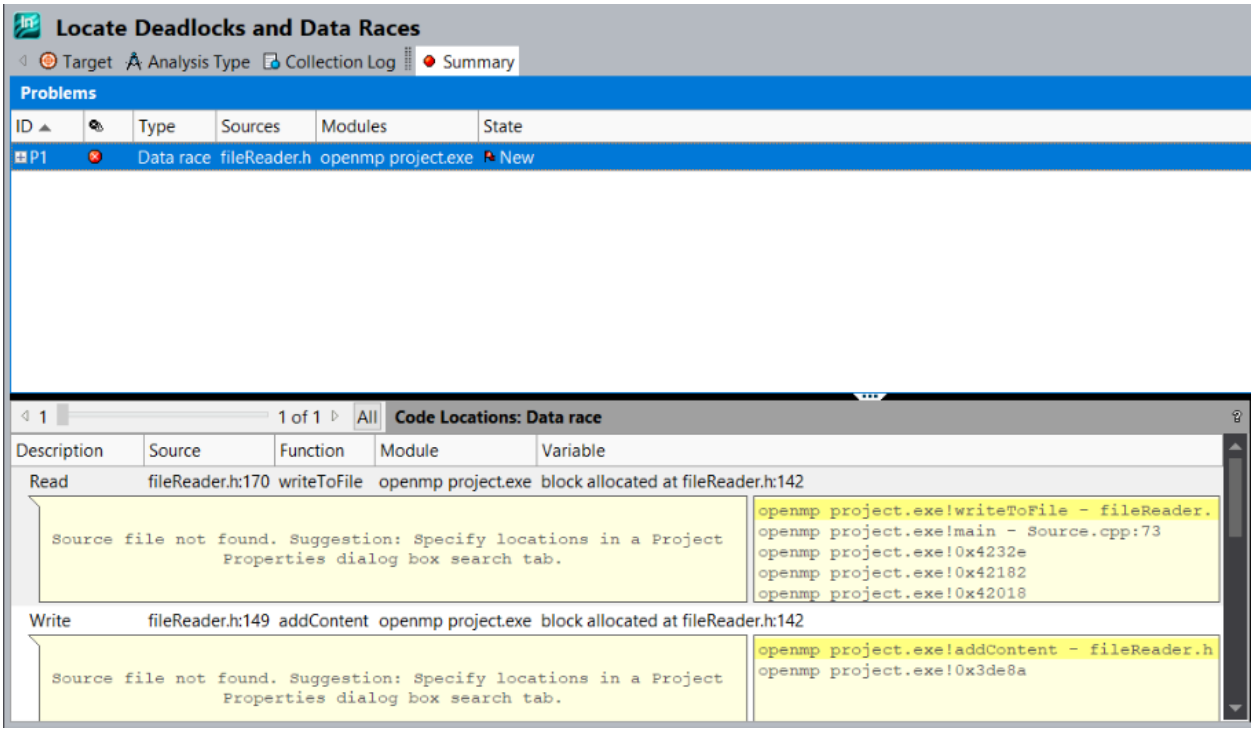
این روش با فرض اینکه تعداد ماتریس زیاد باشند، به خوبی جواب می‌دهد. پیاده‌سازی عملا مانند روش سری است، با این تفاوت که در هر فایل، ماتریس ها بصورت موازی حساب می‌شوند.

پیاده سازی:

با چند تغییر جزیی در کلاس‌های استفاده شده برای اداپت کردن با حالت موازی، به نتیجه می‌رسیم.

```
    // Do this for all files in the directory data_in/
    for (i = 0; i < files.size(); i++) {
        // LOCAL VARIABLES FOR EACH FILE
        // The determinants of matrices in this files are stored here line by line
        string result = "";
        // The output address to write the final result in it
        string outputAddress    = files[i];
        // Read this file from data_in/filename.txt
        FileReader fileReader = FileReader(files[i]);
        // Write output in data_out/filename.txt
        FileWriter fileWriter =
        FileWriter(outputAddress.replace(
        outputAddress.find("_in"), sizeof("_in") - 1, "_out"), fileReader.getNLines());
        // Do this for each matrix in this file
        omp_set_nested(0);
        #pragma omp parallel for schedule(dynamic) num_threads(cores)
        for (j = 0; j < fileReader.getNLines(); j++) {
            float   det;        // Determinant of this matrix
            int*    m;          // This matrix
            // Build a matrix Parser
            MatrixParser matrixParser = MatrixParser();

            // Parse string of numbers to matrix
            m = matrixParser.parseMatrix(fileReader.getLines(j));

            // Build a dataset from m
            DataSet dataset = DataSet(matrixParser.getSize());
            dataset.fillDataSet(m);
            free(m);

            // Calculate determinant using customized LU decomposition
            det = determinantLU(&dataset);

            // Store det to result string
            fileWriter.addContent(j, det);
        }

        // write result string for this file
        fileWriter.writeToFile();
    }
```

با تحلیل Thread Debugger به نتیجه‌ی زیر می‌رسیم:



این مشکل بخاطر این است که تابع addContent می‌تواند همزمان به فایل توسط چند نخ به fileWriter پاسخ دترمینان را حساب کند. برای حل این مشکل کافی‌است قسمت هایلایت شده‌ی کد صفحه‌ی قبل را درون critical قرار دهیم.

<u>زمان:</u> این کد به میانگین زمانی ۶۰ میلی‌ثانیه می‌رسد.

$$Speedup\ 1 = \frac{153}{60} = 2.55$$

پروفایل:

Elapsed Time: 0.119s

CPU Time:	0.360s
Instructions Retired:	924,000,000
Microarchitecture Usage:	42.4% of Pipeline Slots
CPI Rate:	1.324
Total Thread Count:	11
Paused Time:	0s

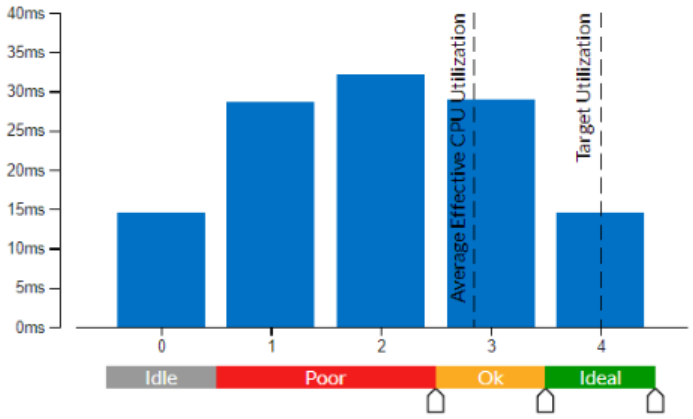
Explore Additional Insights

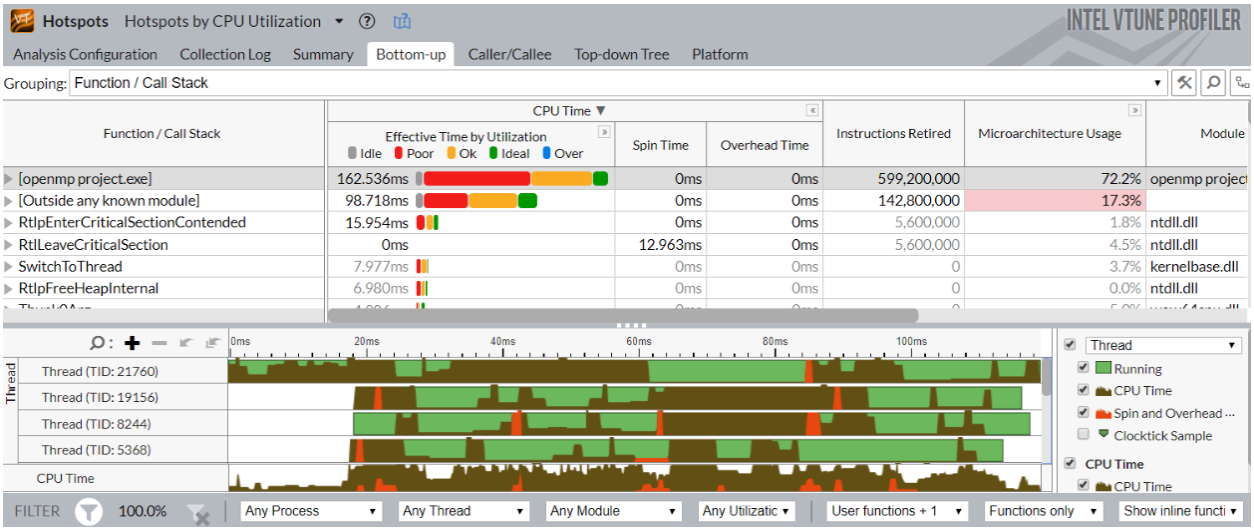
Parallelism : 71.2% (2.850 out of 4 logical CPUs)

Use Threading to explore more opportunities to increase parallelism in your application.

Microarchitecture Usage : 42.4%

Use Microarchitecture Exploration to explore how efficiently your application runs on the used hardware.





این روش هر چه تعداد ماتریس‌های موجود در یک فایل بیشتر باشد، بازدهی بهتری دارد.

روش دوم: هر نخ یک فایل را می‌خواند، حساب می‌کند و جواب را می‌نویسد.

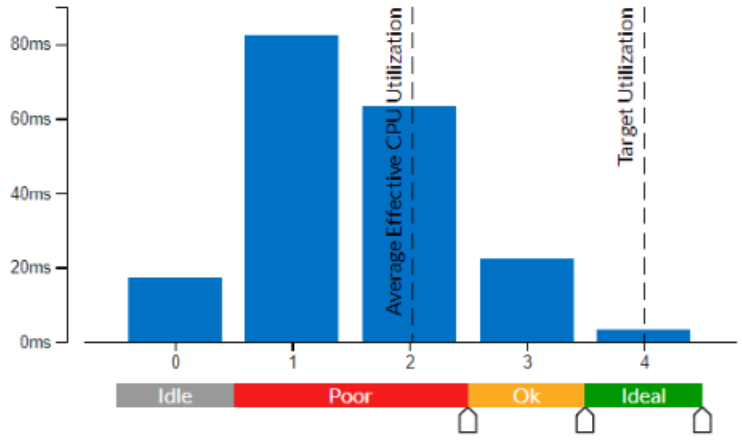
این روش وقتی بازدهی خوبی دارد که تعداد فایل‌ها زیاد و ماتریس‌های درون آن کوچک باشند. این روش هم عملا اجرای برنامه‌ی سری است که هر برنامه فقط یک فایل را می‌خواند و این برنامه‌ها بصورت موازی اجرا می‌شوند.

پیاده‌سازی: مانند روش قبل است، با این تفاوت که pragma قبل از حلقه‌ی اول قرار دارد و i و j در آن private است.

زمان: با اجرای این کد به زمان میانگین ۱۱۴ میلی‌ثانیه می‌رسیم.

$$Speedup\ 2 = \frac{153}{114} = 1.34$$

پروفایل:



روش سوم: یک نخ فایل‌ها را می‌خواند و با الگوی تولیدکننده و مصرف‌کننده دترمینان‌ها را حل می‌کنیم.

در این روش یک نخ فقط مسئول خواندن و پارس کردن ماتریس هاست و بقیه فقط دترمینان را بدست می-آورند.(Source Codes/ PARALLEL 3)

<u>پیاده‌سازی</u>: نخ صفر، مسئول خواندن و پارس کردن ماتریس‌هاست و بعد از اینکه همه‌ی فایل‌ها را خواند و صف وظایف را آماده کرد، بصورت مرتب پوشه‌ی data_out را چک می‌کند و وقتی تعداد فایل‌های نوشته شده برابر تعداد فایل‌های ورودی شد، قفل مصرف کننده‌ها را باز می‌کند.

مصرف کننده‌ها هم دترمینان را حساب می‌کنند و به لیست نتایج می‌ریزند. نتایج هر فایل وقتی کامل شدند، خودشان نتایج را روی فایل مورد نظر می‌ریزند.

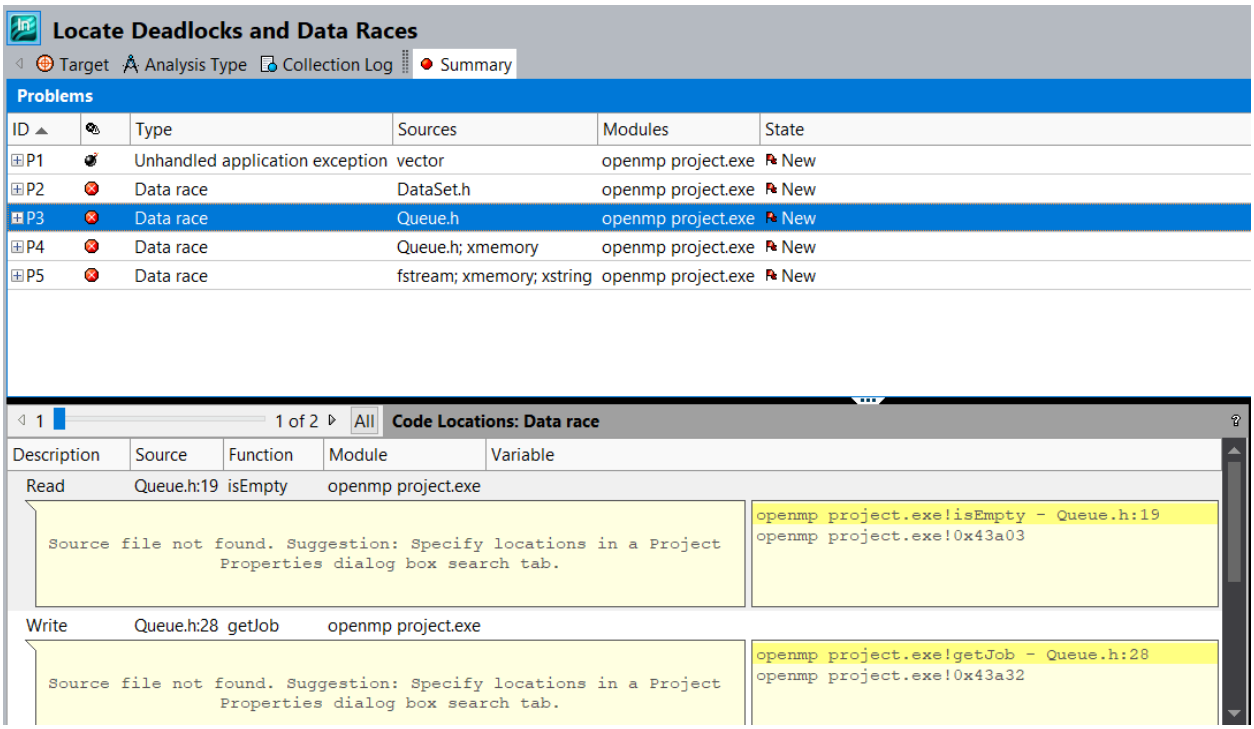
<u>زمان:</u> این روش به میانگین زمان ۶۷ میلی‌ثانیه رسید.

$$Speedup\ 3 = \frac{153}{67} = 2.28$$

```
#pragma omp parallel num_threads(cores)
    {
        int id = omp_get_thread_num();

        if (id == 0) {
            for (i = 0; i < filesNumber; i++) {
                j = 0;
                fstream newfile;
                newfile.open(files[i], ios::in);

                j = 0;
                string matrixLine;
                while (getline(newfile, matrixLine)) {
                    int* m = mp.parseMatrix(matrixLine);
                    jobs.addJob(DataSet(mp.getSize(), m, i, j));
                    j++;
                }

                newfile.close();
            }
        }

        if(id != 0) {
            while (flag) {
                if (!jobs.isEmpty()) {
                    DataSet d = jobs.getJob();
                    float det = determinantLU(&d);
                    results[d.getFile()].addResult(d.getLine(), det);
                }
            }
        }
```

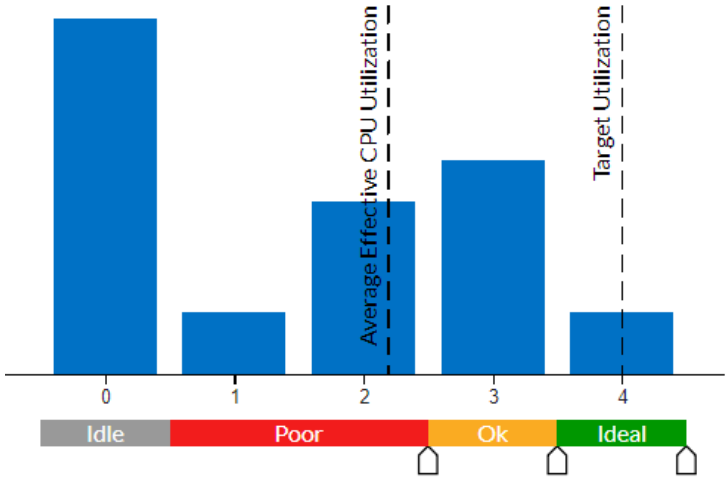
```
        if (id == 0) {
              FolderReader nf = FolderReader("data_out/");
              while (true) {
                    if (filesWritten >= filesNumber) break;
                    nf.readFolder();
                    filesWritten = nf.size();
              }
              flag = false;
        }
  }
```

در این روش، با تست کیس معمول حدود ۲ درصد جواب‌ها غلط است. به عبارت دقیق تر در فایل خروجی عدد ۶۹.۶۹ چاپ می‌شود که مقدار پیش‌فرض جواب هاست. پس حتما شرایط مسابقه داریم.

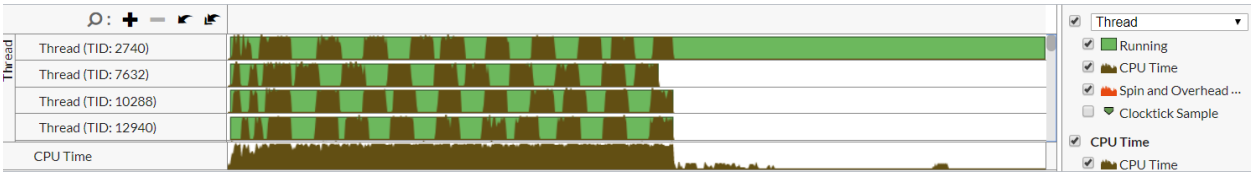


همانطور که مشخص است در صف چند شرط مسابقه داریم که برای حل آن‌ها باید از omp flush کمک بگیریم. ولی چون جواب با همین race ها از روش قبلی سریعتر نیست این روش را ادامه نمی‌دهیم.

پروفایل:



زیاد بودن زمان idle بخاطر این است که initialize کردن تولیدکننده و مصرف کننده طول می‌کشد. می‌توان برای کاهش این زمان، initialize کردن را هم بصورت موازی انجام داد که در کد ضمیمه شده انجام شده.



بعد از تغییرات جدید، کامپایل debug خطای ucrtbased.pdb داد و بعد از آن قادر نبودم پروفایل کنم.

روش چهارم: چند نخ فایل‌ها را می‌خوانند و کارت گرافیکی آن‌ها را حل می‌کنند.

در این روش از GPU هم کمک می‌گیریم؛ کرنل اولیه به شکل زیر است:

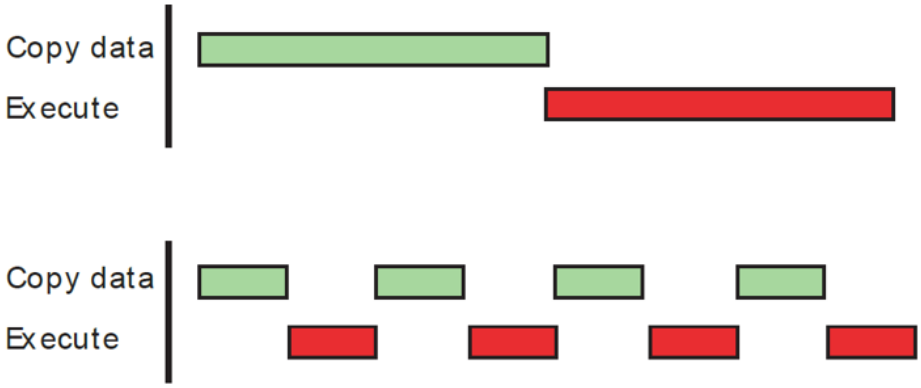
```
__global__ void addKernel(float* u, int col, float* det)
{
	if (col == ARRAYSIZE - 1) {
		det[0] *= u[ARRAYSIZE * ARRAYSIZE - 1]; // Handle last element
		return;
	}
	int i = threadIdx.x + col;
	int j = blockIdx.x + col + 1;
	det[0] *= u[col * ARRAYSIZE + col];
	float z = u[j * ARRAYSIZE + col] / u[col * ARRAYSIZE + col];

	u[j * ARRAYSIZE + i] -= z * u[col * ARRAYSIZE + i];
}
```

که تابعی که آن را صدا می‌زند به صورت زیر است:

```
float caldet(float* u) {
    cudaError_t cudaStatus;

    float* det;
    float* dev_u = 0;
    float* dev_det = 0;
    int i, j;

    det = (float*)malloc(sizeof(float));
    det[0] = 1.f;

    // Allocate GPU Buffers
    cudaStatus = cudaMalloc((void**)&dev_u, ARRAYSIZE * ARRAYSIZE * sizeof(float));
    cudaStatus = cudaMalloc((void**)&dev_det, sizeof(float));


    // Copy input vectors from host memory to GPU buffers.
    cudaStatus = cudaMemcpy(dev_u, u,
                 ARRAYSIZE * ARRAYSIZE * sizeof(float), cudaMemcpyHostToDevice);
    cudaStatus = cudaMemcpy(dev_det, det, sizeof(float), cudaMemcpyHostToDevice);

    // Call the kernel for each column
    for (i = 0; i < ARRAYSIZE; i++) {
        int blockDim = ARRAYSIZE - 1 - i;
        int threadDim = ARRAYSIZE - i;
        if (blockDim == 0) {
            blockDim++;            // Handle last column/element
        }
        addKernel << <blockDim, threadDim >> > (dev_u, i, dev_det);
    }

    cudaStatus = cudaMemcpy(det, dev_det, sizeof(float), cudaMemcpyDeviceToHost);

    cudaFree(dev_u);
    return det[0];
}
```

همانطور که در کد دیده می‌شود نیازی به کپی کردن ماتریس U به حافظه‌ی هاست نیست و این سرعت را کمی بهبود می‌بخشد.

زمان انتقال فایل‌ها خیلی زیاد است. با فقط یکبار کپی کردن کل داده‌ها هم تغییر چندانی رخ نمی‌دهد:



راه حل‌ها:

۱. یک راه **همپوشان کردن کپی و اجرای کرنل**هاست که ایده‌ی ساده‌ی آن می‌تواند استفاده از دو نخ باشد.
۲. راه دیگر استفاده از **استریم** هاست.
۳. استفاده از **unified Memory** هم سرعت را به شدت کم می‌کند.

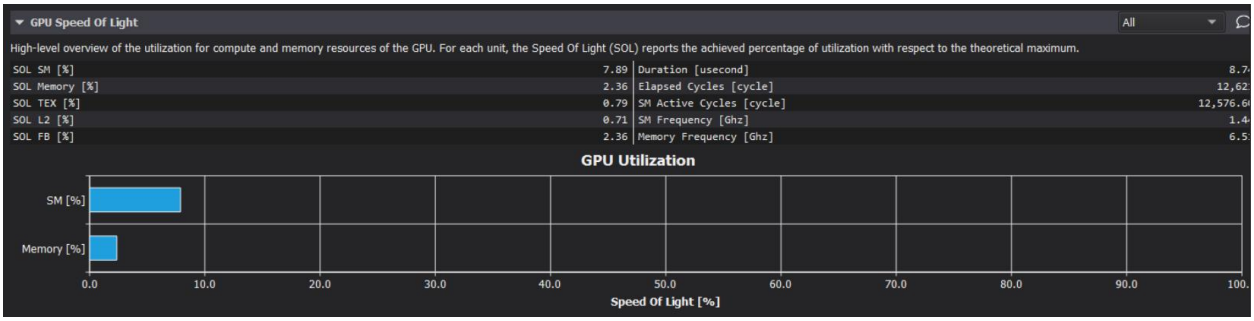
برای مقایسه، الگوریتم سریال را با این الگوریتم روی تعداد مختلف ماتریس با سایز ثابت اجرا می‌کنیم. با این کار می‌توانیم حد آستانه‌ای را پیدا کنیم که GPU قوی‌تر از CPU کار می‌کند.

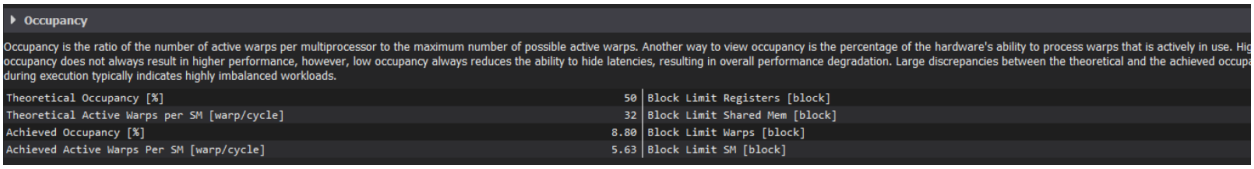
تعداد	اندازه	CPU Single Core Time(ms)	GPU Time(ms)		
			کرنل	کپی داده	کل
200	16	3.33	4.5	110	134
400	16	6.53	10.2	128	158
200	32	16.87	11	98	149
200	64	110	23	126	163

همانطور که دیده می‌شود، سربار کپی داده آنقدر زیاد است که بهتر است از GPU استفاده نکنیم.

برای ماتریس‌های بزرگ (۳۲ و ۶۴) که تعداد بالایی هم دارند، می‌توان از قدرت کارت گرافیکی بهره گرفت. ایده‌ی من این است که در سیستم تولیدکننده، مصرف‌کننده، یک صف برای ماتریس‌های بزرگ ایجاد کنیم (صف۳۲ و صف۶۴) و دو نخ را به این دو اختصاص دهیم که کرنل‌ها را صدا بزنند.

آنالیز Nsight





همانطور که دیده می‌شود، occupancy بسیار کمی داریم و دلیل آن این است که به ازای هر ماتریس، کرنل را به تعداد ستون‌هایش صدا می‌زنیم.

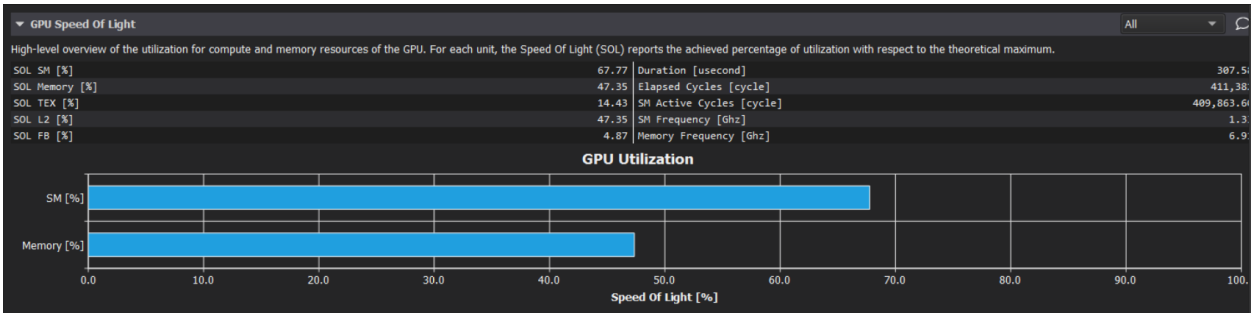
بهبود ۱: با تغییر کد، می‌توانیم فقط یکبار dev_u را کپی کنیم و روی کل ماتریس‌ها فقط به تعداد ستون‌ها کرنل را صدا بزنیم ولی به تعداد ماتریس‌ها بلوک داشته باشیم.(Source Codes/ CUDA)

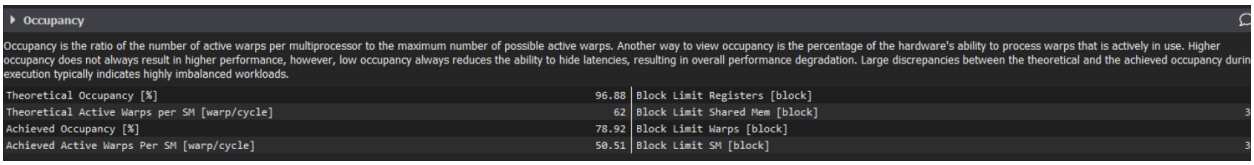
```
#pragma omp parallel for num_threads(devices)
    for (i = 0; i < 1; i++) {
        int threadId = omp_get_thread_num();
        cudaStatus = cudaSetDevice(threadId);
        detHolder = gpuDet(matHolder, i, threadId, matsize);
}
```

کرنل هم به شکل زیر تغییر می‌کند.

```
__global__ void detKernel(float* u, int col, float* det, int size)
{
    int m = blockIdx.x;
    int first = m * size * size;
    if (col == size - 1) {
        det[m] *= u[first + size * size - 1];
        return;
    }
    int i = threadIdx.x + col + 1;
    int j = threadIdx.y + col;
    det[m] *= u[first + col * size + col];
    float z = u[first + i * size + col] / u[first + col * size + col];

    u[first + i * size + j] -= z * u[first + col * size + j];
}
```

آنالیز Nsight





برای اتصال پروژه‌ی CUDA به پروژه‌ی OPENMP به مشکل خوردم و نتوانستم از این به بعد ادامه دهم و هدر .cuh را به پروژه‌ی قبلی اضافه کنم.(`materials/detKernelHeader.cuh`)