آرش حریرپوش
9731505

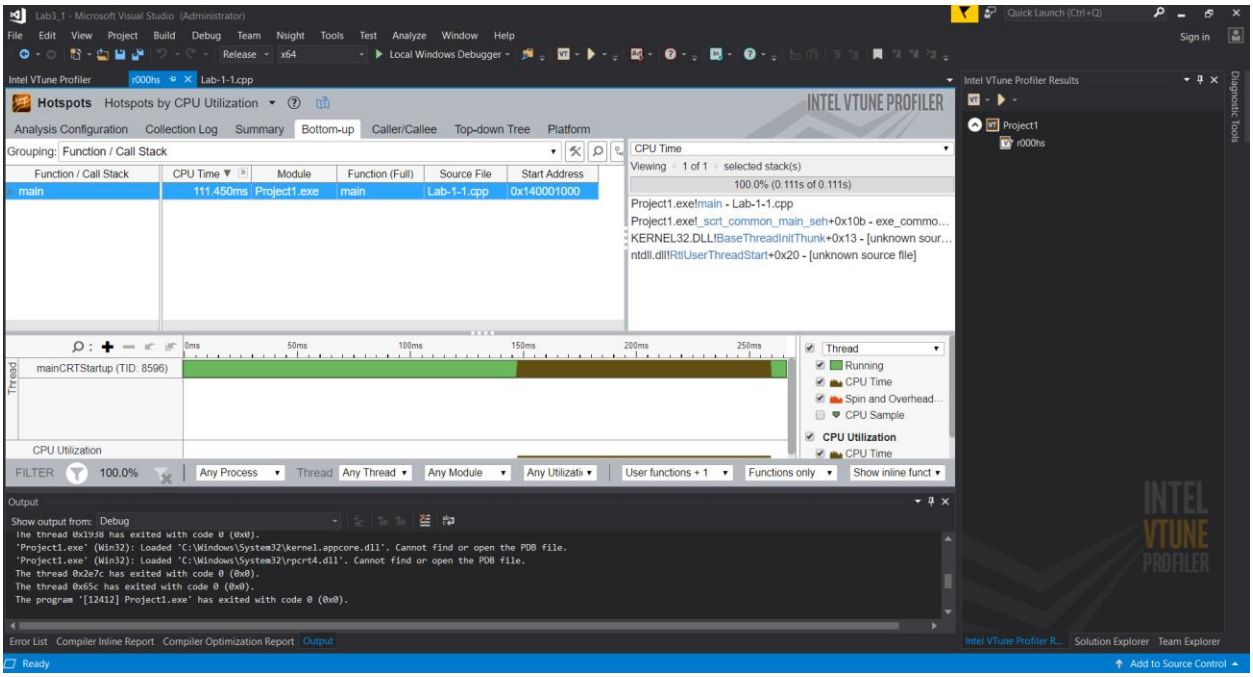
قسمت اول آزمایش

برای اینکه یک برنامه را به صورت اصولی موازی کنیم مراحل زیر را انجام می دهیم:

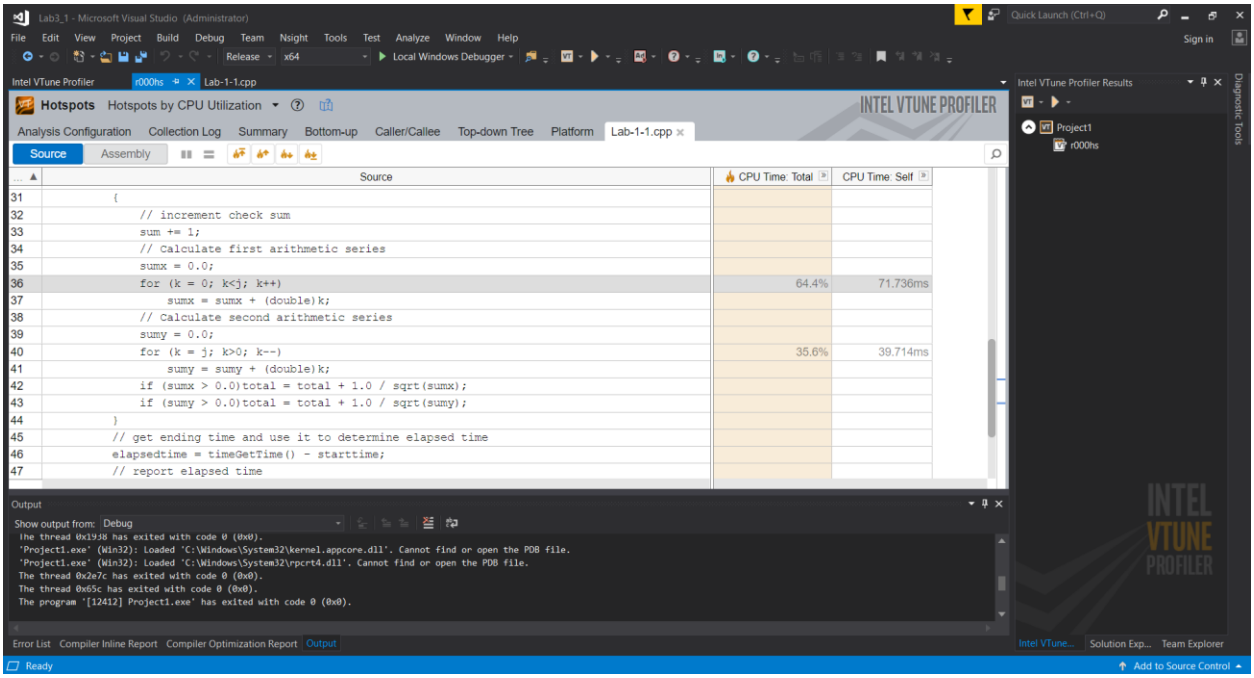
1. برنامه سریال را آنالیز می کنیم تا موقعیت های موازی سازی آن را پیدا کنیم.
2. با استفاده از ساختار های موازی سازی مدل موازی سازی مورد نظر خود را پیاده سازی می کنیم.
3. برنامه را برای خطا های احتمالی ناشی از موازی سازی deug می کنیم.
4. برنامه موازی را تنظیم می کنیم. به گونه ای که سربار های اضافی زیادی نداشته باشیم و کار ها بین نخ ها به صورت تقریبا یکسان تقسیم شده باشد.

1. مرحله آنالیز برنامه سری

در ابتدا برنامه را به کمکVtune Profiler و حالت hotspot به صورت سری آنالیز می کنیم، تا نقاط قابل موازی سازی آن را پیدا کنیم، که نتیجه این آنالیز به صورت زیر می باشد:



شکل1- نتیجه HotSpot analysis به کمک VTune Profiler



شکل2- نمایش hotspots بدست آمده از نتیجه HotSpot analysis به کمک VTune Profiler

از نتایج شکل2، که نقاط hotspots را نشان می دهد به این نتیجه می رسیم که بیشترین زمان اجرای این برنامه متعلق به حلقه های محاسباتی می باشد که ابتدا حلقه خط 36 با 64.4 زمان اجرا بیشترین سهم اجرا و پس از آن حلقه خط 40 با 35.6 درصد از زمان اجرا بیشترین سهم اجرا را دارد؛ بنابراین با موازی کردن این حلقه ها می توانیم برنامه را موازی سازی کنیم.

2. پیاده سازی موازی سازی به کمک OpenMP

سپس در مرحله بعد با توجه به نتایج بدست آمده در شکل2، برنامه را به صورت زیر موازی می کنیم:

1. ابتدا کتابخانه openmp را به صورت زیر به برنامه اضافه می کنیم:

```
#include <omp.h>
```

2. سپس با موازی کردن حلقه خط 31، برنامه را به صورت زیر موازی می کنیم:

```
// Work loop, do some work by looping VERYBIG times

#pragma omp parallel for

 for( int j=0; j<VERYBIG; j++ )

 {
```

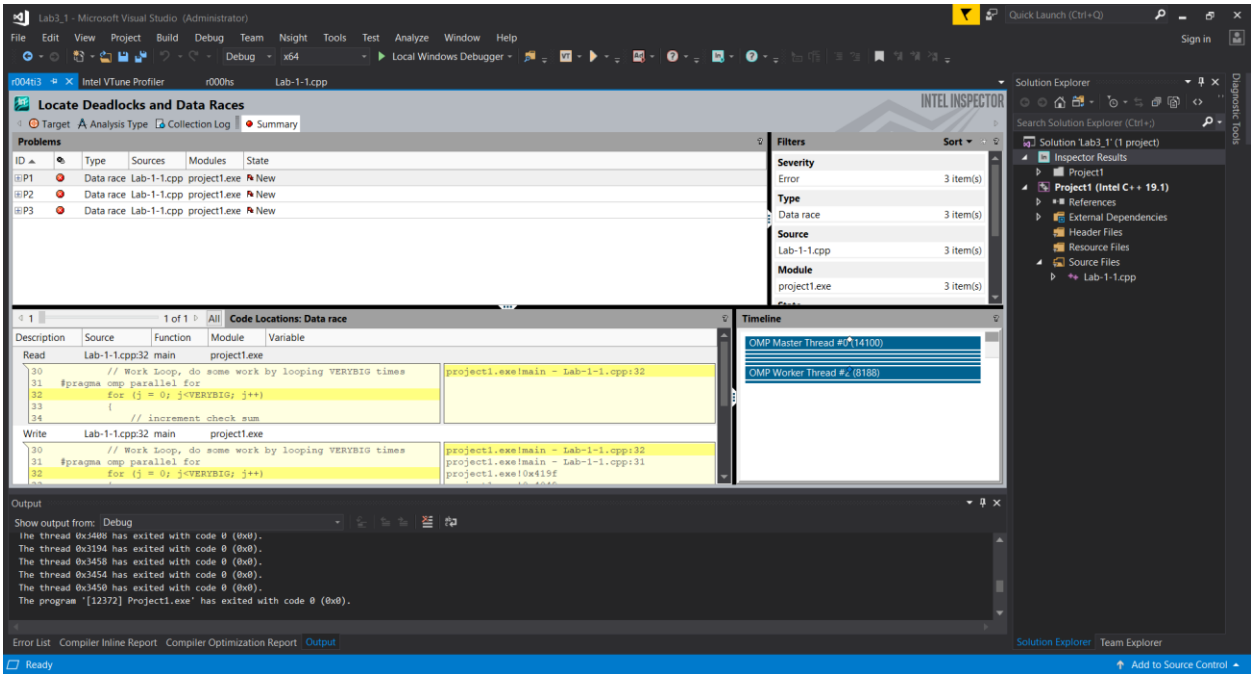
3. Debug و چک کردن برای خطا ها

در مرحله بعد به دلیل اینکه برنامه را موازی کرده ایم و با این کار به دلیل اینکه اجرای برنامه را همروند کرده ایم، ممکن است برنامه به صورت درست اجرا نشود به همین دلیل به کمک مراحل زیر این مشکلات را برطرف می کنیم و سرعت و اجرای برنامه را بهبود می دهیم.

برای پیدا کردن race ها و deadlock ها از Intel Parallel Inspector XE استفاده می کنیم.
برای این کار برنامه را به صورت debug بجای releaseکامپایل می کنیم، به دلیل اینکه در حالت release به دلیل بهبود هایی که در این کامپایل انجام می شود، ممکن است به صورت تصادفی خطا هایی را پنهان کند.

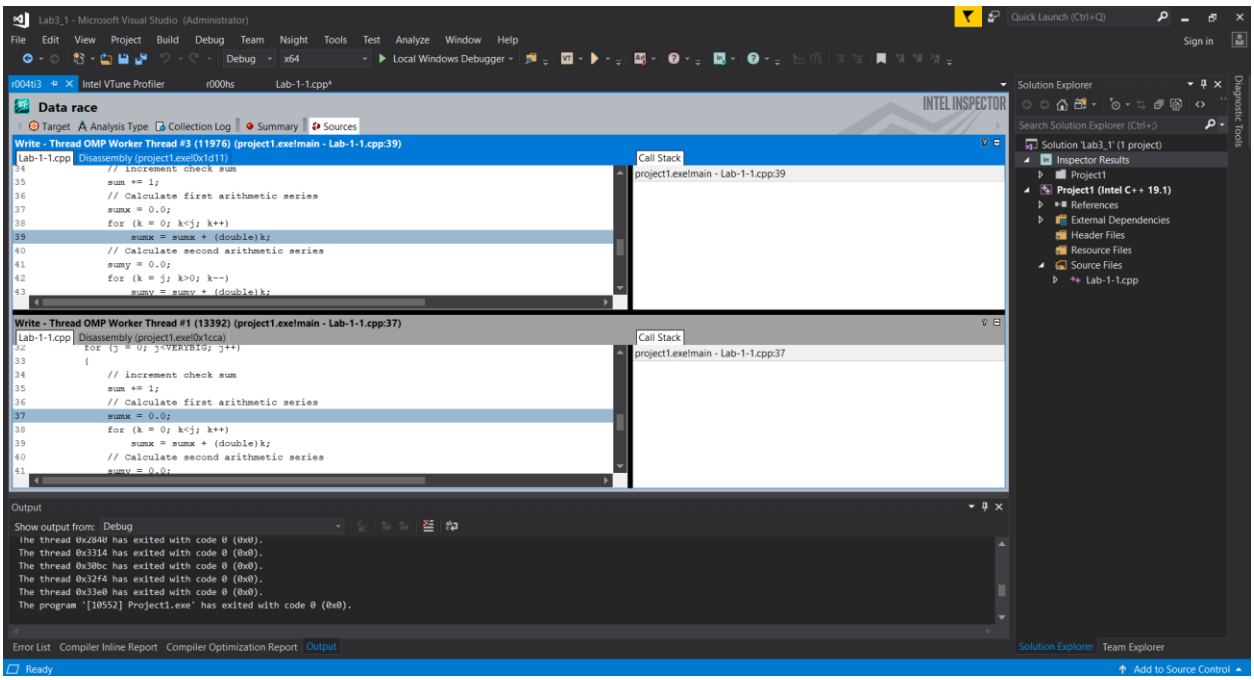
برای پیدا کردن خطا ها مراحل زیر را انجام می دهیم:

1. Configuration را به debug تغییر می دهیم اما آن را rebuild نمی کنیم.
2. به دلیل اینکه Inspector کند است، مقدار VRYBIG را به 1000 تغییر می دهیم، و تکرار حلقه خارجی را به یک کاهش می دهیم.
3. برنامه را rebuild می کنیم.
4. سپس توسط Inspector برنامه را به صورت threading error و حالت Located Deadlocks and Data races آنالیز می کنیم.



شکل3- خلاصه خطا هایی که threading error هایی که توسط آنالیز InspectorXE پیدا شده اند.

پس از انجام مراحل بالا نتیجه همانند شکل3 نمایش داده می شود، با توجه به نتیجه نشان داده شده کد دارای 3 خطا می باشد که به دلیل Data race رخ داده اند، به این صورت که نخ های متفاوت در زمان یکسان می خواهند از یک متغییر بخوانند یا در یک متغییر بنویسند.



شکل4- data race که در کد رخ داده است.

سپس با دیدن خطا ها، که در شکل4 یکی از آن ها نشان داده شده است، در می یابیم که در متغییر های sum، total، sumx، sumy و k، data race رخ داده است.

برای برطرف کردن data race موجود در متغییر های sumx، sumy و k آن ها را برای هر یک از نخ ها آن ها را برای هر نخ private می کنیم؛ این کار باعث می شود که هر نخ یک نسخه از این متغییر ها داشته باشد و هر کدام روی متغییر متناظر خود کار کنند.

همچنان متغییر های sum و total مقدارشان نادرست و به ازای هر جا متفاوت است، که این مشکل نیز به دلیل data race رخ داده است، اما تنها private کردن این متغییر ها کافی نمی باشد، به این دلیل که این متغییر ها باید بین نخ ها shared باشند. برای برطرف کردن این مشکل از reduction استفاده می کنیم.

Reduction به این صورت عمل می کند آن متغییر ها را برای هر نخ private می کند و سپس در نهایت عملیات متناظر را که در این مثال جمع می باشد انجام می دهد، به این صورت که نتیجه هر یک از این متغییر ها را در متغییر shared متناظر آن جمع می کند.

4. تنظیم کردن برنامه OpenMP

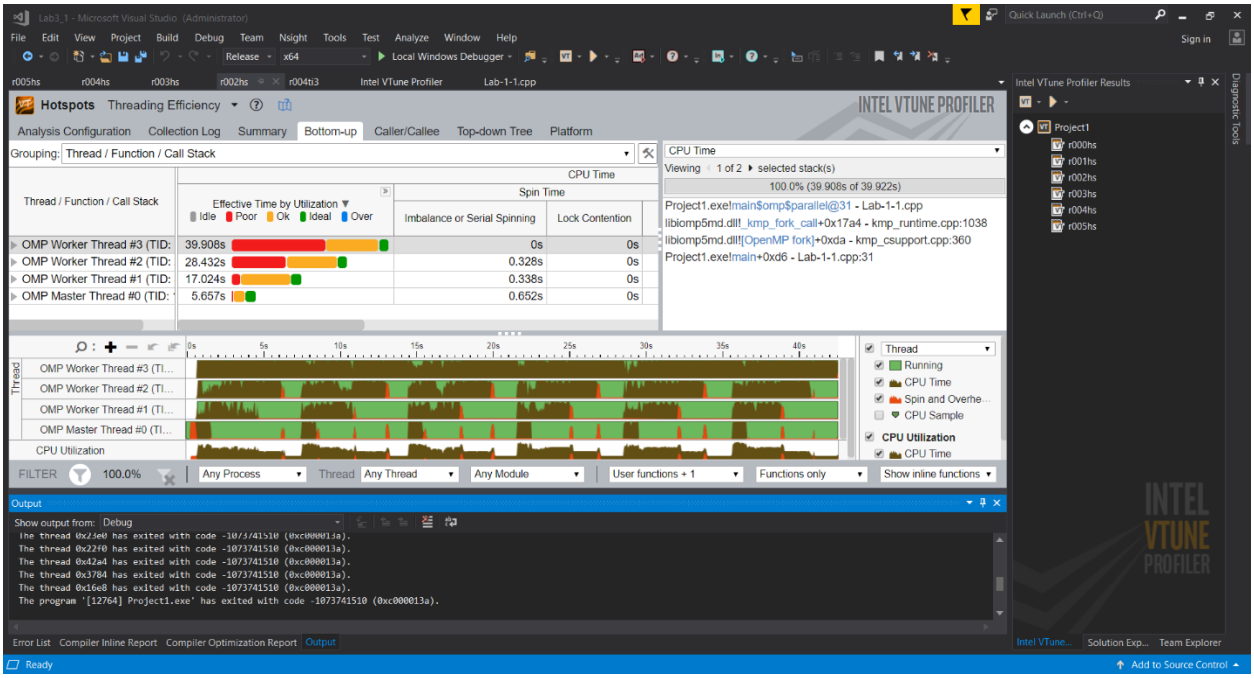
سپس از طریق VTune Profiler و حالت آنالیز Hotspots و سپس بررسی Threading Efficiency میزان کارآیی و همروندی برنامه را بررسی می کنیم.

شکل های زیر یک عملکرد ضعیفی را نمایش می دهند؛ در شکل5، میزان فعالیت نخ ها را مشاهده می کنید، همانطور که از این شکل نیز مشخص است نخ ها به صورت یکسان فعالیت نمی کنند یا به عبارت دیگر load balancing در آن ها وجود ندارد، برای مثال می توان نخ 0 و نخ 3 را با یک دیگر در نظر گرفت که مشاهده می شود نخ 3 بیشترین کار را انجام می دهد در حالی که نخ 0 کمترین کار را انجام می دهد؛ در شکل6، میزان همروندی نخ ها با یک دیگر را مشاهده می کنید. به طور میانگین 2.140 (ضریب همروندی نخ ها برابر 2.140 می باشد) نخ ها به طور همروند با یک دیگر اجرا می شوند، به همین دلیل است که سرعت افزایش پیدا کرده است.

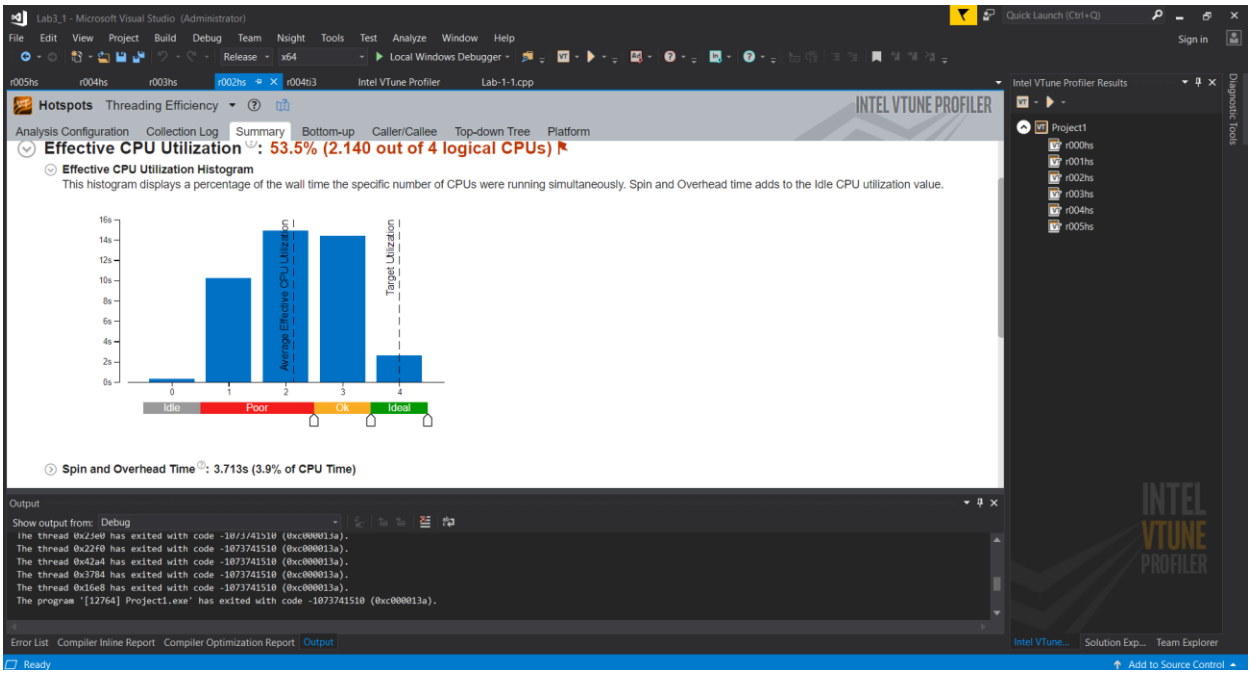
در این برنامه به این دلیل این نحوه موازی سازی به خوبی عمل نکرده است که، در این سری ریاضیاتی با افزایش تعداد تکرار حلقه ها میزان کار آن تکرار نیز افزایش می یابد (میزان کار مورد نیاز برای هر تکرار حلقه با افزایش آن تکرار ها افزایش می یابد)، و عملیات scheduling در OpenMP اجرای برنامه را به صورت یکسان (به اندازه تکرار هایی که بین هر نخ تقسیم می شود chunk size می گویند) بین نخ های موجود تقسیم می کند (هر نخ مسئول بخشی از تکرار ها می شود برای مثال نخ 1 همیشه تکرار های 0 تا 24999 را انجام می دهد)، بنابراین نخ هایی که روی تکرار های بزرگ تر کار می کنند کار بیشتری برای انجام دادن دارند، که به همین دلیل باعث ایجاد unbalanced loading در نخ می شود.

نتایج این تنظیم را در شکل های 5 و 6 مشاهده می کنید.

آرش حریرپوش
9731505



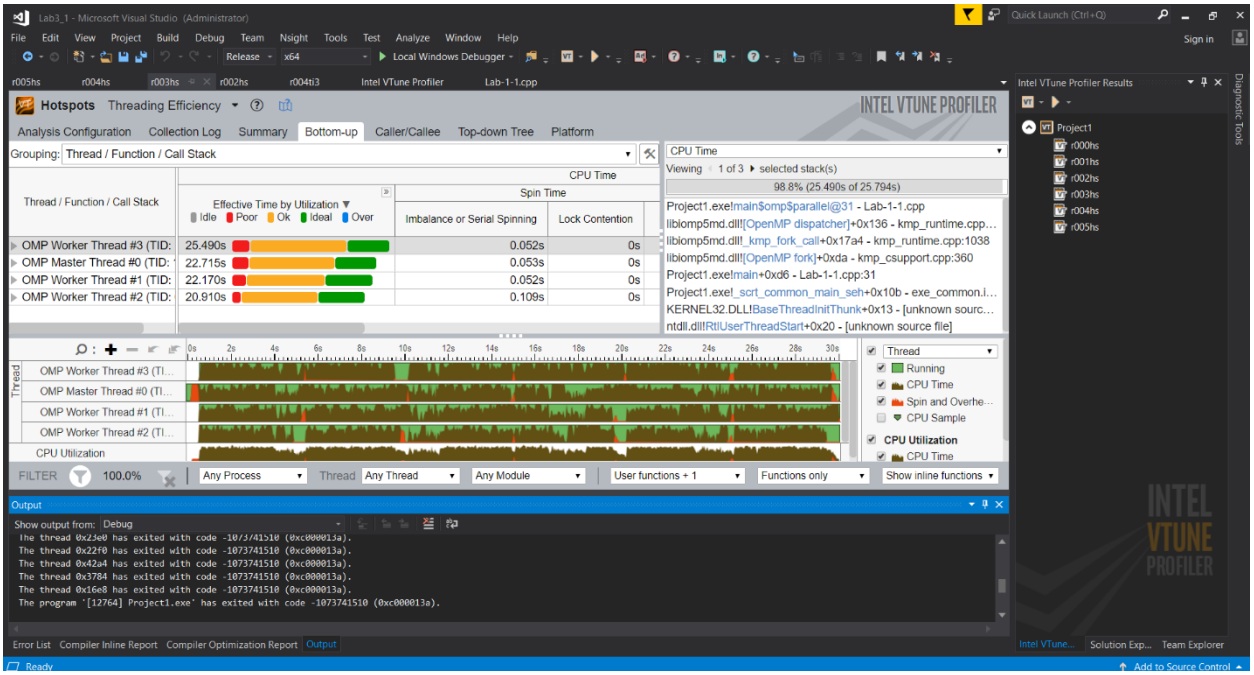
شکل5- اطلاعات نحوه اجرا برنامه زمانی که unbalanced load وجود دارد.



شکل6- نمایش همروندی نخ ها در برنامه OpenMP قبل از تنظیم کردن.

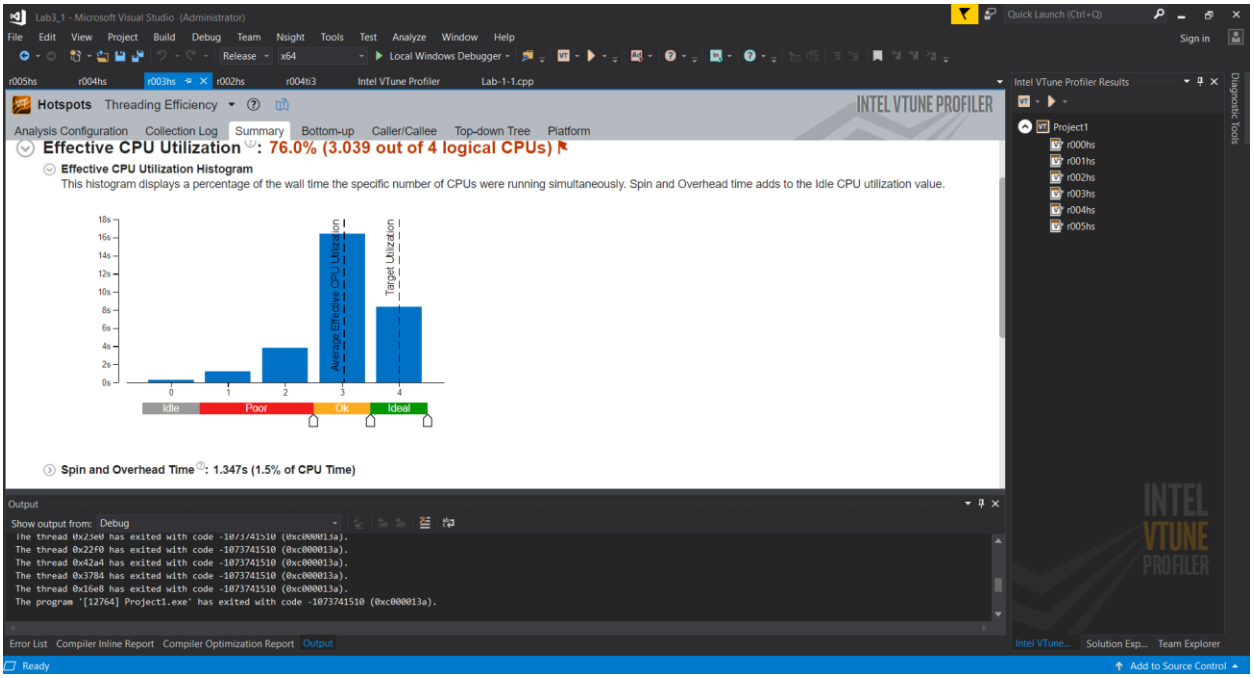
آرش حریرپوش
9731505

سپس برای برطرف کردن این مشکل scheduling را تغییر می دهیم، و برای تنظیم اول chunk size را برابر 2000 می گذاریم، و نوع آن را نیز dynamic می کنیم تا یک نخ با اتمام کار خود به سراغ 2000 تکرار بعدی برود؛ می توانید نتیجه این تنظیم را در شکل های 7 و 8 مشاهده کنید، و همانطور که از این شکل ها می بینید کار تقسیم شده بین نخ ها تقریبا با یک دیگر برابر می باشد و نسبت به حالت قبلی بهبود زیادی پیدا کرده است که به دلیل ایجاد load balancing در کار ها بدست آمده است، و از شکل 10 متوجه می شویم که ضریب همروندی نخ ها نیز به 3.039 از 4 نخ تغییر کرده است، که نسبت به تنظیم قبلی بهبود پیدا کرده است.



شکل7- اطلاعات نحوه اجرا برنامه زمانی که balanced load وجود دارد، اما ایده آل نمی باشد.(chunk size = 2000)

آرش حریرپوش
9731505

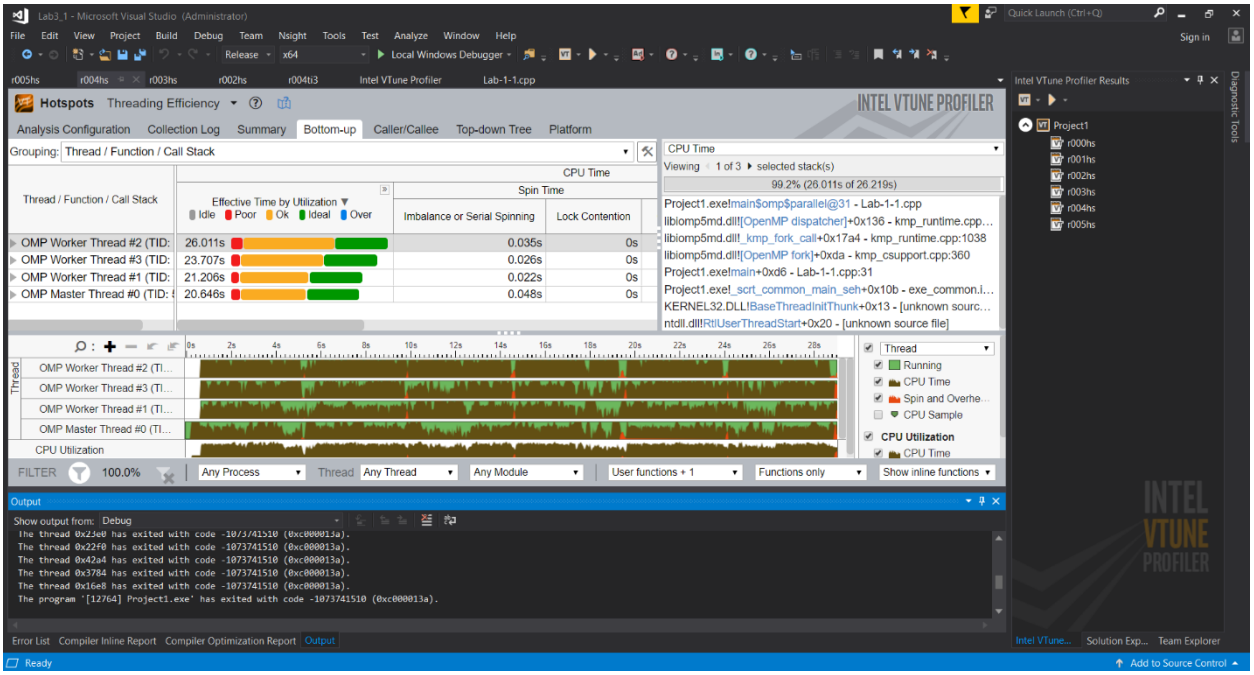


شکل8- اطلاعات همروندی بعد از اولین تنظیم.(chunk size = 2000)

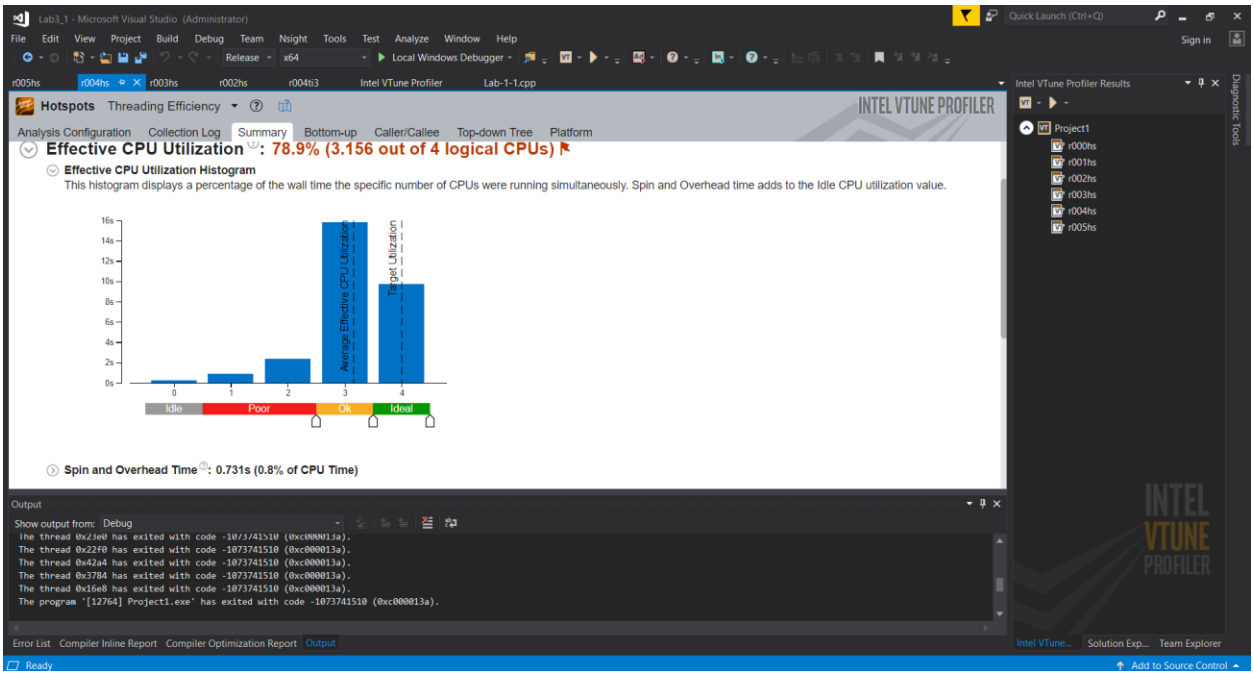
اگرچه با تغییرات مرحله قبل به بهبود خوبی دست پیدا کرده ایم، اما همچنان به حالت ایده آل نرسیده ایم؛ برای بهبود دادن بیشتر و استفاده بهتر از نخ ها مقدار chunk size را به 1000 کاهش می دهیم، با این کار تجزیه را ریز دانه تر کرده ایم، به دلیل اینکه حلقه load balancing ندارد و بار کاری تکرار های بزرگ بیشتر است به همین دلیل ریز دانه کردن باعث بهبود استفاده از نخ ها می شود اما به دلیل اینکه chunk size های خیلی کوچک سربار زیادی دارند باید chunk size مناسب برای این مسئله را بیابیم که هم سربار کمی داشته باشد و هم load balancing را افزایش دهد و میزان کار ها به صورت یک نواخت تری بین نخ ها تقسیم شود، برای محقق کردن این هدف یک بار chunk size را به 1000 که نتایج آن در شکل های 9 و 10 قرار داده شده است و بار دیگر chunk size را به 512 کاهش داده ایم که در شکل های 11 و 12 قرار داده شده است.

همانطور که از شکل های 9 و 10 پیداست، با کاهش chunk size به 1000 برنامه به صورت یکنواخت تری نسبت به مقدار 2000 تقسیم شده است و نخ ها به صورت یکنواخت تری کار می کنند، و میزان ضریب همروندی نخ ها نیز به 3.156 افزایش یافته است (این مقدار یعنی به طور میانگین 3.156 نخ ها همزمان با یک دیگر در حال فعالیت هستند) که اندکی از مقدار تغییر قبلی (chunk size = 2000) بیشتر است.

آرش حریرپوش
9731505



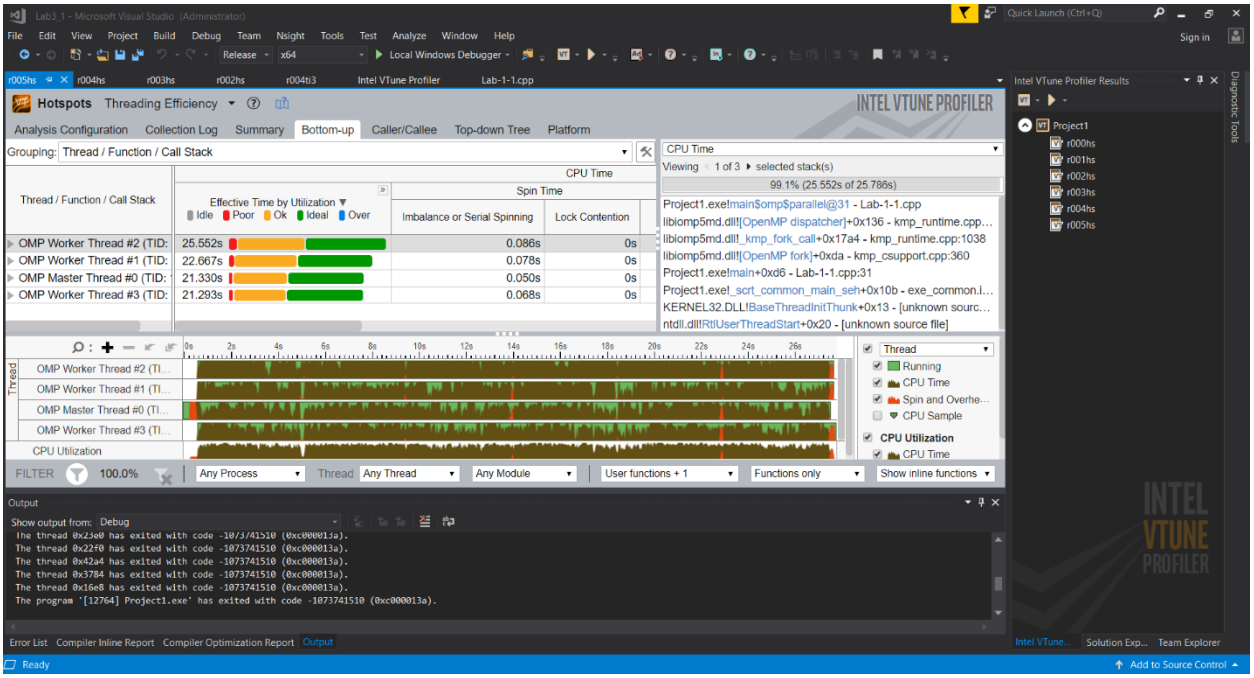
شکل9 - اطلاعات نحوه اجرا برنامه زمانی که balanced load وجود دارد.(chunk size = 1000)



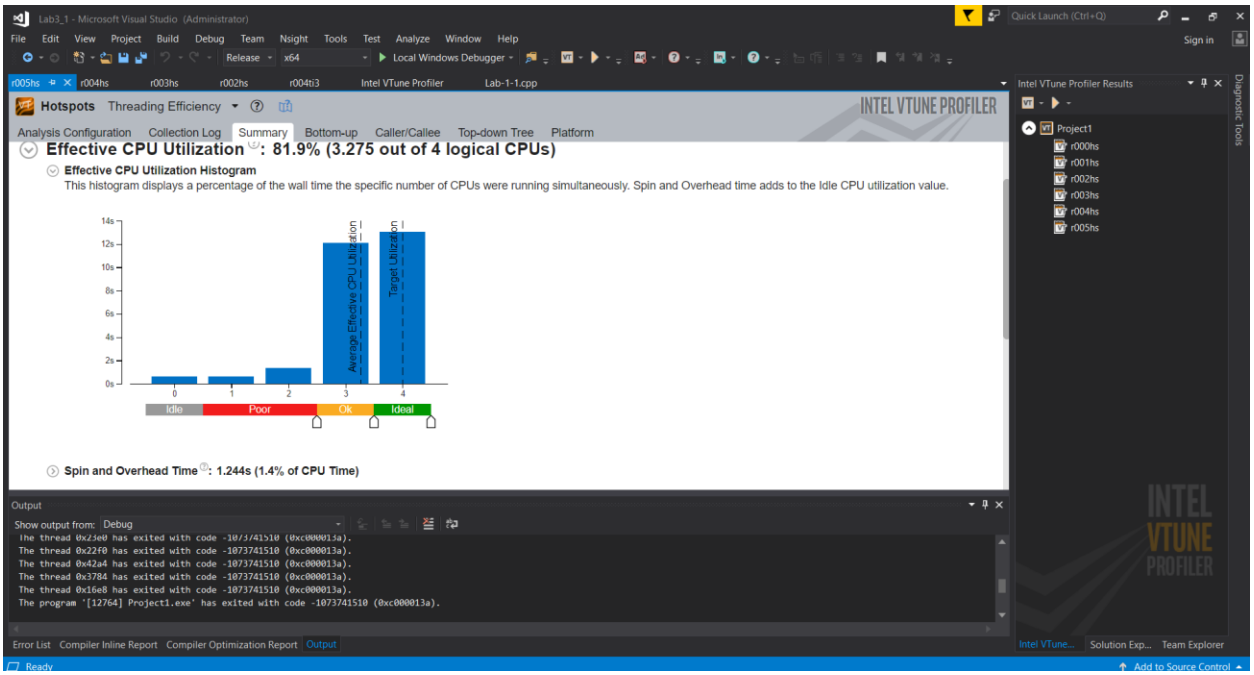
شکل10- اطلاعات همروندی بعد از بهبود تنظیم قبلی.(chunk size = 1000)

همان طور که از شکل های 11 و 12 پیداست، با کاهش chunk size به مقدار 512 کار به صورت یکنواخت تری نسبت به 1000 تقسیم شده و نخ ها یکنواخت تر از حالت قبل کار می کنند، همچنین ضریب همروندی نخ ها نیز به 3.275 افزایش پیدا کرده است که اندکی از حالت قبل بیشتر است.

آرش حریرپوش
9731505



شکل11- اطلاعات نحوه اجرا برنامه زمانی که balanced load وجود دارد.(chunk size = 512)



شکل12- اطلاعات همروندی بعد از تغییر آخرین تنظیم.(chunk size = 512)

با کاهش chunk size تا حد قابل مشخصی استفاده از نخ ها بهینه تر می شود، که با توجه به آزمایش های انجام شده برای این مسئله chunk size برابر 512 نتیجه مطلوبی بدست آورده است و هزینه سر بار آن منطقی و load balancing آن نیز خوب بوده است؛ بنابراین chunk size برابر 512 برای این مسئله مطلوب می باشد.

آرش حریرپوش
9731505

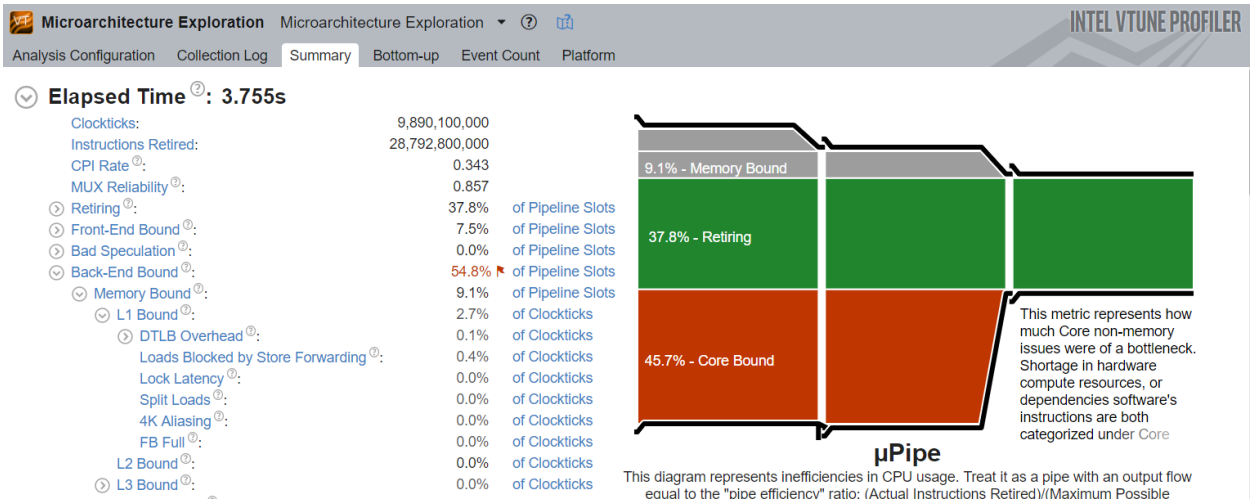
قسمت دوم آزمایش

با توجه به اینکه امکان غیر فعال کردن hyper threading در BIOS موجود نبوده، از روش زیر استفاده کرده ام که عنوان شده است همانند غیر فعال کردن hyper threading می باشد:

1. از طریق Start به تنظیمات "msconfig" می رویم.
2. سپس بخش boot آن را کلیک می کنیم.
3. و بعد از آن بر روی "Advanced Options" کلیک می کنیم.
4. مقدار "Number of procssors" را برابر 4 می گذاریم.
5. سپس دستگاه را restart می کنیم.

(منبع: https://forums.lenovo.com/t5/Gaming-Laptops/no-bios-option-to-disable-hyper-threading/td-p/4430773)

1. فایل کد موردنظر lab1-2.cpp نام دارد که یک جمع ماتریس N در M با دسترسی سطری به درایه های ماتریس است. این کد را با اندازه های بزرگ (برای مثال 8192 در 8192) اجرا کنید و شاخص های بالا را برای آن گزارش کنید.

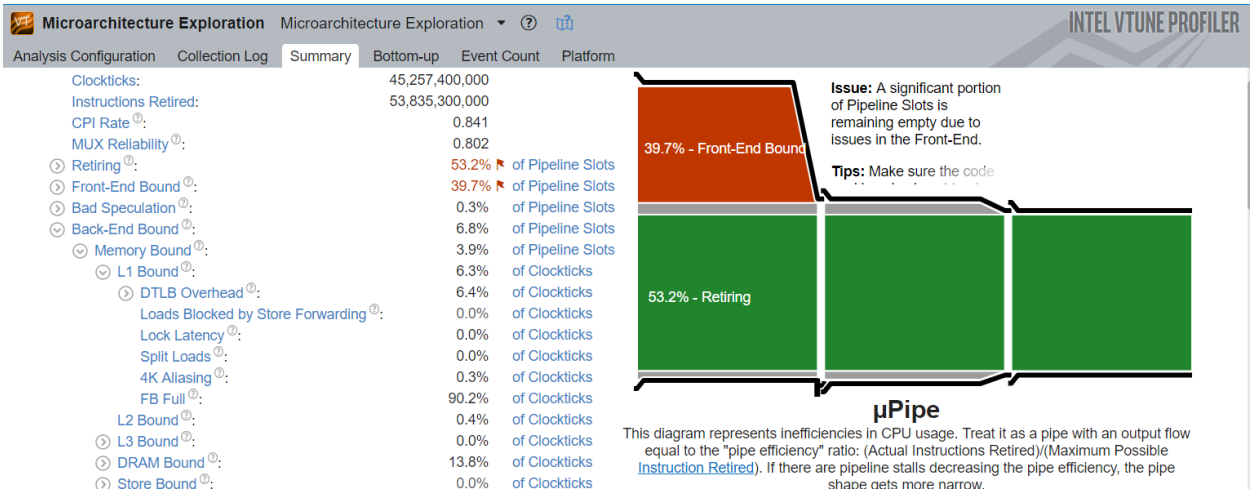


با توجه به اینکه شاخص MUX برابر 0.857 می باشد، بنابراین جواب بدست آمده قابل اعتماد می باشد.

Array Size	L1 Bound	L2 Bound	L3 Bound
(8192, 8192)	2.7%	0.0%	0.0%

آرش حریرپوش
9731505

2. سپس کد را به گونه ای تغییر دهید که دسترسی به درایه های ماتریس به صورت ستونی انجام شود. کد حاصل را با اندازه های بزرگ (برای مثال 8192 در 8192) اجرا کنید و شاخص های بالا را برای آن گزارش کنید.



با توجه به اینکه شاخص MUX برابر 0.802 می باشد، بنابراین جواب بدست آمده قابل اعتماد می باشد.

Array Size	L1 Bound	L2 Bound	L3 Bound
(8192, 8192)	6.3%	0.4%	0.0%

3. آیا اثر این پدیده در ورودی با اندازه های مختلف ثابت است؟ برای اثبات جواب خود، زمان محاسبه ی جمع ماتریس های 16000 در 16000، 4096 در 4096، 2048 در 2048 و 64 در64 در دو حالت یک و دو را به دست آورید.

خیر، اثر این پدیده با افزایش سایز ماتریس ها افزایش می یابد.

Array Size	Column Time	Row Rime
(16000, 16000)	14.982888	0.543504
(4096, 4096)	0.503254	0.033559
(2048, 2048)	0.131389	0.008204
(64, 64)	0.000016	0.000004

با محاسبه زمان مورد نیاز برای جمع ماتریس ها به کمک تابع omp_get_wtime()، به درستی گذاره گفته شده می رسیم که اثر این پدیده با اندازه های مختلف متفاوت است، چراکه هرچه سایز ماتریس بزرگ تر شود در روش ستونی تعداد miss های آن نیز بیشتر می شود به همین دلیل زمان اجرای آن افزایش می یابد.

آرش حریرپوش
9731505

4. درصورتی که حجم مورد استفاده ی هر سطر از حجم هر cache-line کمتر باشد، آیا زمان اجرا و شاخص های معرفی شده برای هردو حالت نزدیک به هم می شود؟

بله، فاصله زمانی و شاخص های معرفی شده بین آنها کمتر می شود اما همچنان شاخص ها و زمان اجرا برای برنامه سری بهتر می باشد، چراکه در آن تعداد Miss ها کمتری رخ می دهد، حتی اگر کل سطر کامل آورده نشود باز نسبت به زمانی که برنامه ستونی باشد بهتر است، برای مثال اگر هر دفعه دو درایه از یک سطر را پر کنیم تعداد miss هایمان نسبت به حالت سری نصف است، به همین دلیل همچنان برنامه سطری بهتر از ستونی عمل می کند، به عبارتی اگر هر دفعه یکی از خانه های cache را به صورت سطری پر کنیم در این صورت با ستونی برابر می شود.