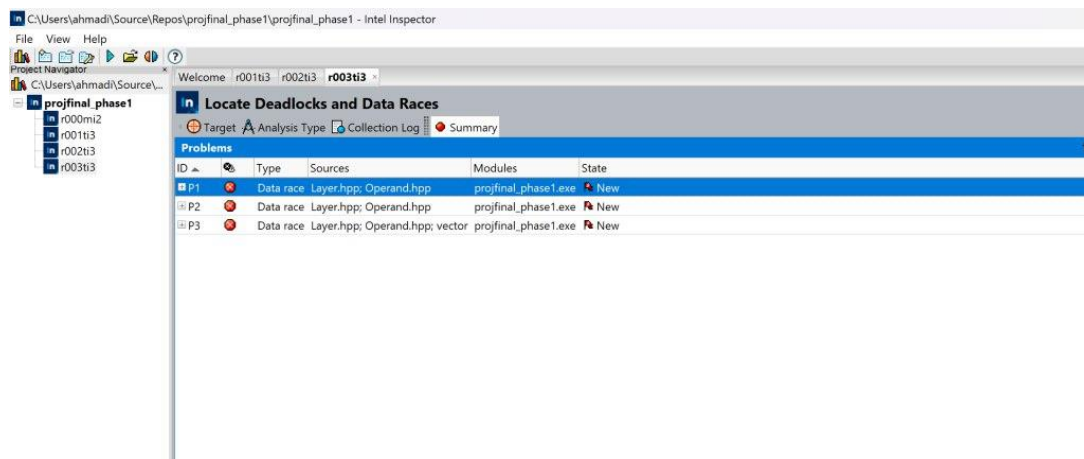


"بسمه تعالی"

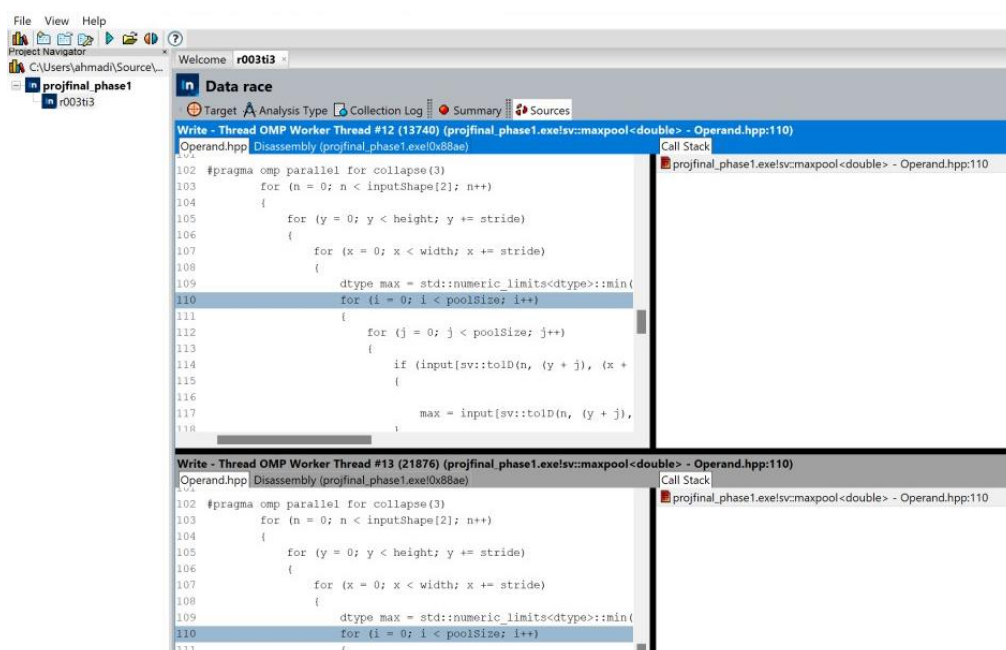
گزارش فاز اول پروژه برنامه‌نویسی چند هسته‌ای - آیدا احمدی پارسا - ۹۹۲۳۰۰۳ - زهرا لطیفی - ۹۹۲۳۰۶۹

▪ بررسی Data Race با ابزار Inspector

می‌دانیم Data Race در OpenMP زمانی رخ می‌دهد که دو یا چند نخ به طور همزمان و بدون همگام‌سازی مناسب به یک مکان حافظه دسترسی داشته باشند و حداقل یکی از دسترسی‌ها عملیات نوشتن باشد. این رخداد می‌تواند منجر به رفتار غیرقابل پیش‌بینی و نتایج نادرست شود. ابزار intel inspector در سه خط از کد داده شده و در فایل Operand.hpp چنین چیزی را گزارش کرد.



۱. `for (i = 0; i < poolSize; i++)`: این حلقه روی بعد `X` تکرار می‌شود. اگر چندین نخ در حال به‌روزرسانی متغیر `max` بدون همگام‌سازی باشند، می‌تواند منجر به Data Race شود.



۲. $\text{for}(j = 0; j < \text{poolSize}; j++)$: به طور مشابه، این حلقه روی بعد y تکرار می‌شود. اگر max توسط چندین نخ به طور همزمان به‌روزرسانی شود، می‌تواند باعث ایجاد Data Race شود.

```

Write - Thread OMP Worker Thread #11 (8068) (projfinal_phase1.exe!sv::maxpool<double> - Operand.hpp:112)
Operand.hpp Disassembly (projfinal_phase1.exe!0x85bc)
1103      for (n = 0; n < inputShape[2]; n++)
1104      {
1105          for (y = 0; y < height; y += stride)
1106          {
1107              for (x = 0; x < width; x += stride)
1108              {
1109                  dtype max = std::numeric_limits<dtype>::min();
1110                  for (i = 0; i < poolSize; i++)
1111                  {
1112                      for (j = 0; j < poolSize; j++)
1113                      {
1114                          if (input[sv::toID(n, (y + j), (x +
1115                          {
1116                              max = input[sv::toID(n, (y + j),
1117                              }
1118                          }
1119                      }
1120                  }
1121              }
1122          }
1123      }

Write - Thread OMP Worker Thread #13 (21876) (projfinal_phase1.exe!sv::maxpool<double> - Operand.hpp:112)
Operand.hpp Disassembly (projfinal_phase1.exe!0x85bc)
1103      for (n = 0; n < inputShape[2]; n++)
1104      {
1105          for (y = 0; y < height; y += stride)
1106          {
1107              for (x = 0; x < width; x += stride)
1108              {
1109                  dtype max = std::numeric_limits<dtype>::min();
1110                  for (i = 0; i < poolSize; i++)
1111                  {
1112                      for (j = 0; j < poolSize; j++)
1113                      {

```

۳. $\text{out}[\text{idx}++] = \text{max}$: این خط حداکثر مقدار یافت شده را به خروجی اختصاص می‌دهد. اگر idx به درستی در بین نخ‌ها همگام‌سازی نشده باشد، یا اگر max توسط چندین نخ قبل از این تخصیص به‌روزرسانی شود، می‌تواند منجر به Data Race شود.

```

Write - Thread OMP Worker Thread #4 (27064) (projfinal_phase1.exe!sv::maxpool<double> - Operand.hpp:122)
Operand.hpp Disassembly (projfinal_phase1.exe!0x88b8)
1113      {
1114          if (input[sv::toID(n, (y + j), (x +
1115          {
1116              max = input[sv::toID(n, (y + j),
1117              }
1118          }
1119      }
1120      }
1121      }
1122      out[idx++] = max;
1123      }
1124      }
1125      }
1126      output = out;
1127      }
1128      }
1129      #ifdef BENCHMARK
1130      MAC = 0;

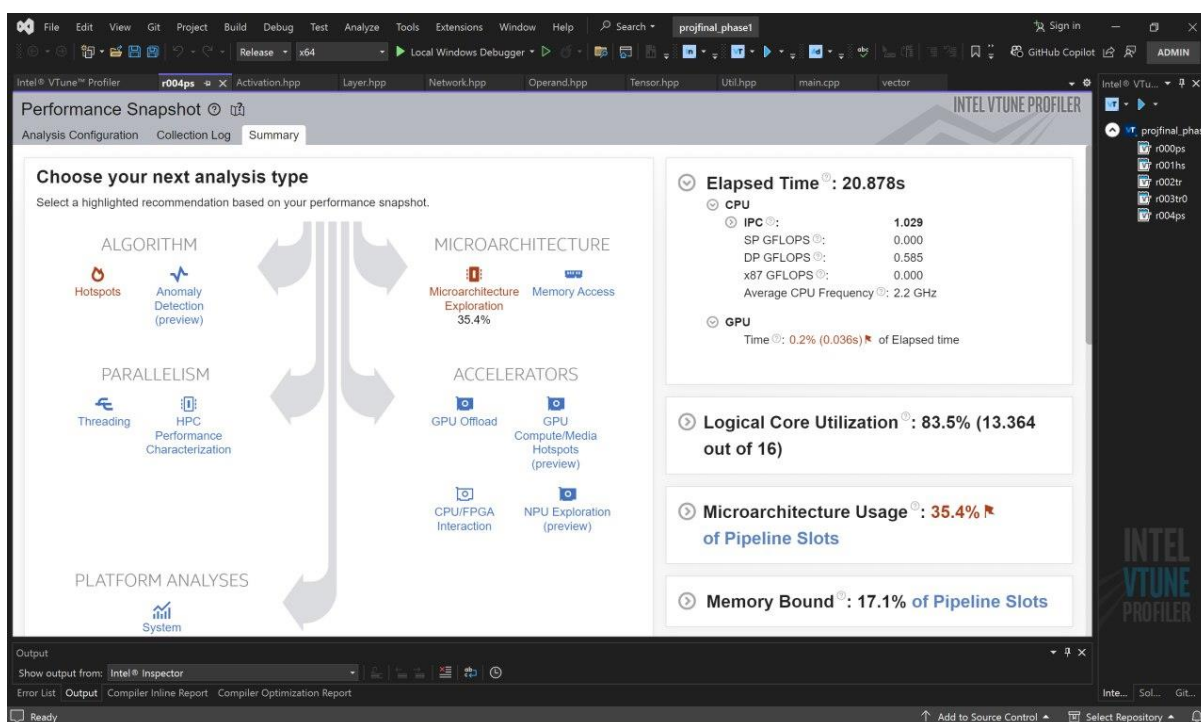
Read - Thread OMP Worker Thread #15 (25076) (projfinal_phase1.exe!operator[] - Operand.hpp:122)
Operand.hpp Disassembly (projfinal_phase1.exe!0x8573)
1113      {
1114          if (input[sv::toID(n, (y + j), (x +
1115          {
1116              max = input[sv::toID(n, (y + j),
1117              }
1118          }
1119      }
1120      }
1121      }
1122      out[idx++] = max;
1123      }

```

از جمله اقداماتی که برای حل این مشکلات می‌توان به کار برد استفاده از `pragma omp atomic#` برای به‌روزرسانی متغیرهای مشترک مانند max ، همگام‌سازی idx به درستی در بین نخ‌ها با استفاده از یک بند `reduction` یا یک عملیات اتمی است.

■ استفاده از VTune و اجرای آنالیز performance snapshot

اولین نوع از آنالیز که با این ابزار انجام دادیم، performance snapshot بود.



Elapsed Time: کل زمان صرف شده برای تکمیل کار که بر حسب ثانیه اندازه گیری می شود برابر با ۲۰.۸۷۸ ثانیه گزارش شد.

IPC (دستورالعمل در هر چرخه): پارامتری برای اندازه گیری کارایی CPU که نشان می دهد چند دستورالعمل در هر سیکل کلاک اجرا می شود برابر با ۱.۰۲۹ گزارش شده است. مقدار IPC بالاتر به این معنی است که CPU در اجرای دستورالعمل ها کارآمدتر است. اگر IPC کمتر از ۱ باشد، نشان می دهد که CPU زمان بیشتری را صرف انتظار داده ها از حافظه یا منابع دیگر می کند، که نشان می دهد برنامه محدود به حافظه است. برعکس، IPC بیشتر از ۱ نشان می دهد که CPU قادر است بیش از یک دستور را در هر چرخه اجرا کند که برای پردازنده های مدرن معمول است.

SP GFLOPS و DP GFLOPS: عملیات ممیز شناور با دقت single و double در هر ثانیه. تعداد عملیات ممیز شناوری که CPU می تواند در ثانیه انجام دهد را نشان می دهند. به ترتیب برابر با ۰ و ۰.۵۸۵ گزارش شده اند.

x87 GFLOPS: استاندارد ممیز شناور قدیمی تر در محاسبات، همچنین تعداد عملیات در ثانیه را نشان می دهد و برابر با ۰ گزارش شده است.

متوسط فرکانس CPU: میانگین سرعتی که CPU در حین انجام کار با آن کار می کند است و بر حسب گیگاهرتز اندازه گیری می شود و برابر با ۲.۲ گیگاهرتز گزارش شده است.

GPU Time درصد زمانی که GPU استفاده شده است نشان می دهد که چه مقدار از کار برای پردازش موازی به GPU محول شده است.

این پارامترها با درک نحوه استفاده از بخش‌های مختلف سیستم، به شناسایی گلوگاه‌ها و بهینه‌سازی عملکرد کمک می‌کنند. به عنوان مثال، حال که استفاده از GPU کم است، نشان دهنده آن است که می‌توان وظایف بیشتری را برای بهبود عملکرد کلی به GPU محول کرد. یا مثلاً اینکه اگر IPC پایین باشد، می‌تواند نشان دهنده ناکارآمدی در نحوه پردازش دستورالعمل‌ها توسط CPU باشد.

⌵ Memory Bound[Ⓢ]: 17.1% of Pipeline Slots

⌵ Performance-core (P-core):

⌵ Memory Bound[Ⓢ]: 17.1% of Pipeline Slots

Cache Bound[Ⓢ]: 31.3% of Clockticks

DRAM Bound[Ⓢ]: 0.9% of Clockticks

⌵ Efficient-core (E-core):

⌵ Memory Bound[Ⓢ]: 19.4% of Clockticks

Cache Bound[Ⓢ]: 3.3% of Clockticks

DRAM Bound[Ⓢ]: 1.5% of Clockticks

**N/A is applied to metrics with undefined value. There is no data to calculate the metric.*

Memory Bound: زمانی اتفاق می‌افتد که عملکرد برنامه توسط سرعت زیرسیستم حافظه محدود شود. اگر برنامه‌ای تعداد دسترسی‌های حافظه زیادی داشته باشد، به خصوص اگر کند باشد یا حافظه پنهان را از دست بدهد، می‌تواند منجر به وضعیتی شود که CPU زمان بیشتری را نسبت به اجرای دستورالعمل‌ها در انتظار اطلاعات از حافظه صرف می‌کند. این مسئله اغلب با نرخ بالای از miss حافظه پنهان و زمان دسترسی طولانی به حافظه نشان داده می‌شود. اگر برنامه‌ای محدود به حافظه باشد، به این معنی است که بهبود الگوهای دسترسی به حافظه یا افزایش پهنای باند حافظه می‌تواند منجر به عملکرد بهتر شود.

این متریک درصد اشغال شدن شکاف‌های خط لوله پردازنده را به دلیل تاخیر در دسترسی به حافظه نشان می‌دهد. درصد بالا نشان می‌دهد که برنامه زمان زیادی را صرف انتظار برای دریافت داده از حافظه می‌کند، که می‌تواند نشان دهد که بهینه‌سازی الگوهای دسترسی به حافظه یا بهبود استفاده از حافظه پنهان ممکن است باعث بهبود عملکرد شود. برابر ۱۷.۱ درصد برای P-core و ۱۹.۴ درصد برای E-core گزارش شده.

Cache Bound: این قسمت از متریک محدودیت حافظه، درصد زمان صرف شده برای fetch داده‌ها از کش را نشان می‌دهد. درصد بالای آن نشان می‌دهد که ممکن است فرصت‌هایی برای بهینه‌سازی کد برای استفاده بهتر از کش وجود داشته باشد. برابر ۳۱.۳ درصد برای P-core و ۳.۳ درصد برای E-core گزارش شده که عدد بالایی برای P-core است.

DRAM Bound: نشان دهنده درصد زمان صرف شده برای fetch داده‌ها از DRAM است. درصد بالای آن می‌تواند به گلوگاه‌های بالقوه در سرعت دسترسی به حافظه اشاره کند. برابر ۰.۹ درصد برای P-core و ۱.۵ درصد برای E-core گزارش شده.

هسته عملکرد (P-core) و هسته کارآمد (E-core): انواع مختلف هسته‌های CPU، با هسته‌های P به طور معمول قوی‌تر و هسته‌های E با انرژی کارآمدتر هستند.

ترکیب این معیارها می‌تواند بینش‌هایی را درباره گلوگاه‌های بالقوه در عملکرد سیستم ارائه دهد و راهنمایی کند که بهینه‌سازی‌ها می‌توانند بیشترین تأثیر را داشته باشند. به عنوان مثال، اگر متریک Memory Bound بالا باشد، ممکن است بهینه‌سازی ساختارهای داده یا الگوریتم‌ها برای کاهش زمان دسترسی به حافظه مفید باشد.

▼ Logical Core Utilization ⓘ: 83.5% (13.364 out of 16)
Physical Core Utilization ⓘ: 71.3% (8.551 out of 12) ▲

Logical Core Utilization: این متریک درصد هسته‌های Logical که به طور فعال استفاده می‌شوند را نشان می‌دهد. نرخ استفاده بالا نشان می‌دهد که بخش قابل توجهی از قدرت پردازش CPU در حال استفاده است. برابر ۸۳.۵ درصد به معنای ۱۳.۳۶۴ از ۱۶ هسته گزارش شده.

Physical Core Utilization: این متریک درصد هسته‌های فیزیکی در حال استفاده را نشان می‌دهد. توجه به این نکته مهم است که CPUهای مدرن می‌توانند چندین هسته Logical در هر هسته فیزیکی داشته باشند، بنابراین این معیار نمای دقیق‌تری از استفاده از هسته ارائه می‌دهد. برابر ۷۱.۳ درصد به معنای ۸.۵۵۱ از ۱۲ گزارش شده است.

این معیارها مرتبط هستند زیرا به شناسایی میزان استفاده از منابع CPU کمک می‌کنند. اگر استفاده از هر یک کم باشد، نشان می‌دهد که فرصت‌هایی برای بهینه‌سازی برنامه برای استفاده بهتر از قدرت پردازش موجود وجود دارد. برعکس، اگر زیاد باشد، می‌تواند نشان دهد که برنامه محدود به CPU است و ممکن است بتواند از موازی‌سازی یا سایر بهبودهای عملکرد بهره‌مند شود.

▼ GPU Active Time ⓘ: 0.2% ▲
GPU Utilization when Busy ⓘ: 4.5% ▲
▼ EU State ⓘ:
Active ⓘ: 4.5%
Stalled ⓘ: 18.8%
Idle ⓘ: 76.7% ▲
Occupancy ⓘ: 11.4% ▲ of peak value

زمان فعال GPU: این معیار درصد زمانی را که GPU به طور فعال taskها را پردازش می‌کند نشان می‌دهد. افزایش این معیار نشان می‌دهد که GPU بیش از گذشته مورد استفاده قرار می‌گیرد. برابر ۰.۲ درصد گزارش شده.

GPU Utilization When Busy: نشان دهنده درصد زمانی است که GPU در هنگام فعال بودن مشغول است. درصد بالاتر به این معنی است که GPU زمانی که بیکار نیست زمان بیشتری را برای انجام وظایف صرف می‌کند. برابر ۴.۵ درصد گزارش شده.

EU State: فعال: درصد واحدهای اجرایی (EU) که به طور فعال وظایف را پردازش می‌کنند.

Stalled: درصدی از EU که منتظر داده‌ها یا منابع هستند که می‌تواند نشان دهنده گلوگاه‌ها باشد. ۱۸.۸ درصد گزارش شده.

Idle: درصدی از EU که در حال حاضر وظایف را پردازش نمی‌کند و برای استفاده در دسترس هستند. برابر ۷۶.۷ درصد گزارش شده.

Occupancy: این متریک نشان می‌دهد که چه مقدار از پتانسیل GPU در حال استفاده است که به صورت درصدی از حداکثر ظرفیت بیان می‌شود. نرخ Occupancy پایین می‌تواند نشان دهد که فرصت‌هایی برای بهینه‌سازی برنامه برای استفاده بهتر از منابع GPU وجود دارد. برابر ۱۱.۴ درصد گزارش شده.

Microarchitecture Usage: 35.4%

of Pipeline Slots

Performance-core (P-core):

Retiring	40.7%	of Pipeline Slots
Front-End Bound	10.1%	of Pipeline Slots
Bad Speculation	0.8%	of Pipeline Slots
Back-End Bound	48.3%	of Pipeline Slots
Memory Bound	17.1%	of Pipeline Slots
Core Bound	31.3%	of Pipeline Slots

Efficient-core (E-core):

Retiring	30.0%	of Pipeline Slots
Front-End Bound	5.7%	of Pipeline Slots
Bad Speculation	2.0%	of Pipeline Slots
Back-End Bound	62.3%	of Pipeline Slots
Core Bound	42.8%	of Clockticks
Memory Bound	19.4%	of Clockticks
Back-End Bound Auxiliary	62.3%	of Pipeline Slots
Resource Bound	62.3%	of Pipeline Slots

*N/A is applied to metrics with undefined value. There is no data to calculate the metric.

Microarchitecture Usage: این متریک استفاده کلی از خط لوله Microarchitecture را نشان می‌دهد. درصد کمتر آن نشان می‌دهد که در نحوه استفاده از منابع سیستم جا برای بهبود وجود دارد. برابر ۳۵.۴ درصد گزارش شده.

Retiring: این پارامتر درصدی از اسلات‌های خط لوله را نشان می‌دهد که توسط دستورالعمل‌هایی که در مرحله بازنشستگی هستند اشغال شده‌اند (تکمیل شده و به رجیسترها یا حافظه بازگردانده می‌شوند).

Front-End Bound: درصد شکاف‌های خط لوله را نشان می‌دهد که توسط دستورالعمل‌هایی اشغال شده‌اند که منتظر داده‌ها از قسمت جلویی خط لوله هستند، که شامل مراحل fetch دستورالعمل و decode است.

Back-End Bound: نشان دهنده درصد شکاف‌های خط لوله است که توسط دستورالعمل‌هایی اشغال شده‌اند که منتظر داده‌ها از قسمت پشتی خط لوله هستند که شامل مراحل اجرا و write-back است.

Core Bound: نشان دهنده درصد شکاف‌های خط لوله است که توسط دستورالعمل‌هایی که منتظر داده‌های هسته‌های دیگر هستند اشغال شده است. می‌تواند مشکلاتی را در ارتباط بین هسته‌ای یا همگام سازی مطرح کند.

Vectorization: 0.0%

of Packed FP Operations

Instruction Mix:

SP FLOPs	0.0%	of uOps
Packed	3.2%	from SP FP
128-bit	3.2%	from SP FP
256-bit	0.0%	from SP FP
Scalar	96.8%	from SP FP
DP FLOPs	2.6%	of uOps
Packed	0.0%	from DP FP
128-bit	0.0%	from DP FP
256-bit	0.0%	from DP FP
Scalar	100.0%	from DP FP
x87 FLOPs	0.0%	of uOps
Non-FP	97.4%	of uOps

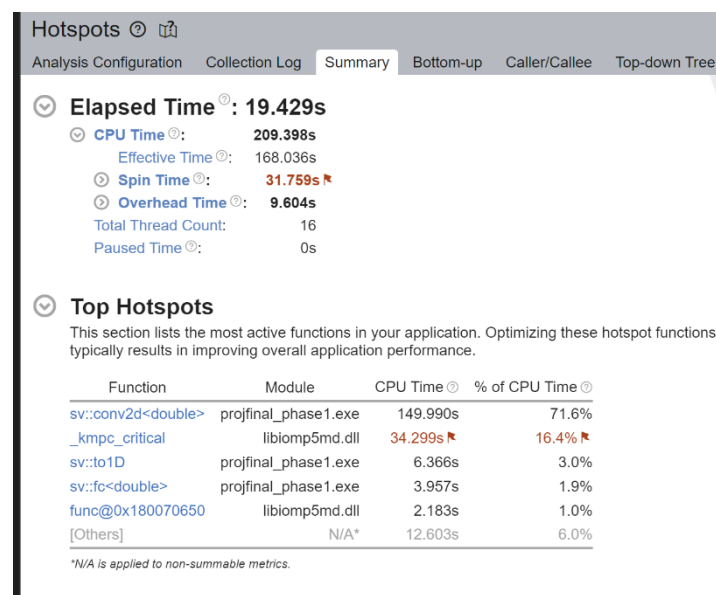
Metrics were collected from Big Cores only.

Vectorization: این متریک درصد عملیات ممیز شناور را نشان می‌دهد که با استفاده از دستورالعمل‌های برداری انجام می‌شود که می‌تواند چندین نقطه داده را در یک دستورالعمل واحد پردازش کند. درصد برداری بالاتر نشان دهنده استفاده بهتر از قابلیت‌های پردازش برداری CPU است.

Packed FP Operations: اینها عملیات ممیز شناور هستند که در یک دستورالعمل بسته بندی می‌شوند و امکان پردازش کارآمدتر را فراهم می‌کنند. درصد عملیات **Scalar FP**، **Packed FP**، **۱۲۸-bit Packed FP** و **۲۵۶-bit Packed FP** ترکیبی از انواع مختلف عملیات ممیز شناور را نشان می‌دهد.

- استفاده از ظرفیت برداری: این متریک نشان می‌دهد که چقدر از ظرفیت پردازش برداری CPU در حال استفاده است. این به ظرفیت‌های ۱۲۸ بیتی و ۲۵۶ بیتی تقسیم می‌شود که منعکس کننده اندازه‌های مختلف ثبات‌های برداری موجود در CPU است.

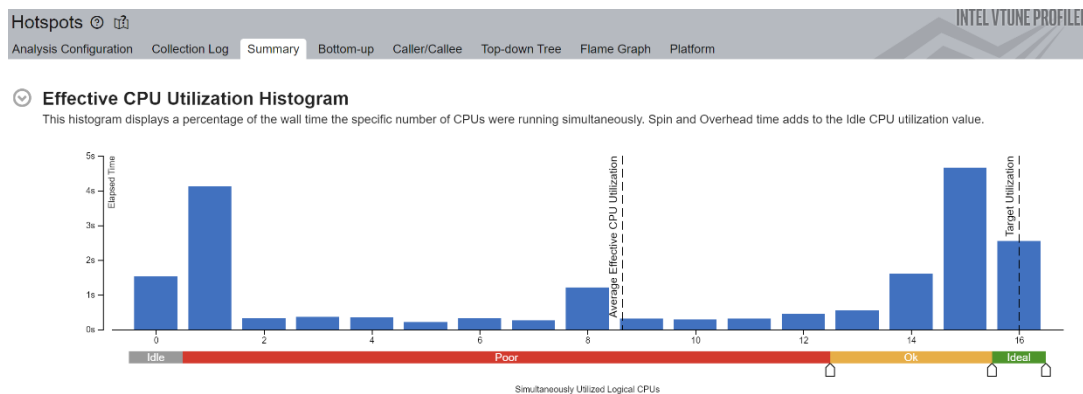
▪ آنالیز Hotspot



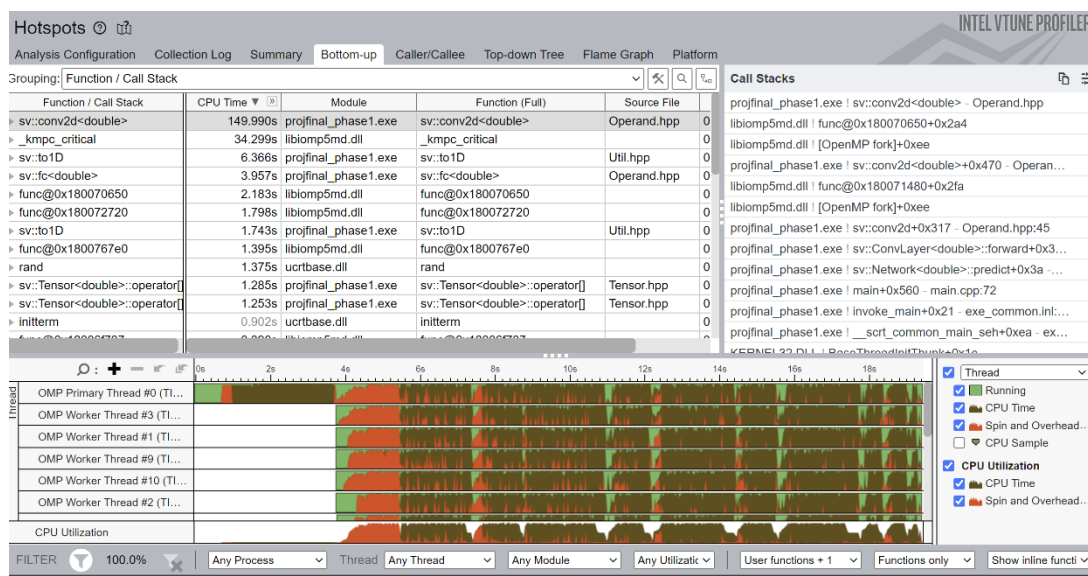
با استفاده از اطلاعات فوق، می‌توان توابع زمان‌بر را شناسایی کرد. به عنوان مثال یکی از زمان‌برترین توابع، تابع **conv2D** است که در فایل هدر **operand** به کار گرفته شده است و در بخش امتیازی به اصلاح این بخش پرداخته‌ایم.

نمودار استفاده از CPU: این نمودار نشان می‌دهد که چه مقدار از ظرفیت CPU در طول زمان استفاده می‌شود. اگر استفاده از CPU به طور مداوم بالا باشد، نشان می‌دهد که برنامه محدود به CPU است، به این معنی که CPU گلوگاه در فرآیند است. برعکس، اگر استفاده از CPU کم باشد، می‌تواند به این معنی باشد که برنامه I/O-Bound است یا منتظر تکمیل عملیات ورودی/خروجی است.

استفاده از CPU برای مناطق زمان‌بر: دانستن اینکه کدام بخش از کد بیشترین مصرف CPU را دارد می‌تواند به شناسایی گلوگاه‌های عملکرد کمک کند. این مناطق جایی هستند که تلاش‌های بهینه‌سازی باید در آن متمرکز شود. استفاده زیاد از CPU در بخش‌های خاص ممکن است نشان دهنده الگوریتم‌های ناکارآمد، حلقه‌های بهینه‌نشده یا سایر بخش‌هایی باشد که می‌تواند از بازسازی کد بهره‌بردار.



تصویر فوق نشان می‌دهد که مصرف CPU در بیشتر اوقات در حالت poor قرار دارد. در نتیجه کد I/O Bound است.



در تصویر فوق مجدداً مشاهده می‌شود که overhead تابع conv2D از سایر توابع و بخش‌های کد بیشتر است. همچنین مشاهده می‌شود که هر تابع توسط کدام نخ در حال اجرا است.

همچنین می‌توانیم به طور جزئی مشاهده کنیم که دقیقاً کدام بخش از تابع بیشترین زمان را مصرف می‌کند.

به عنوان مثال در تابع conv2D دستورات زمان‌بر به شرح زیر می‌باشند.

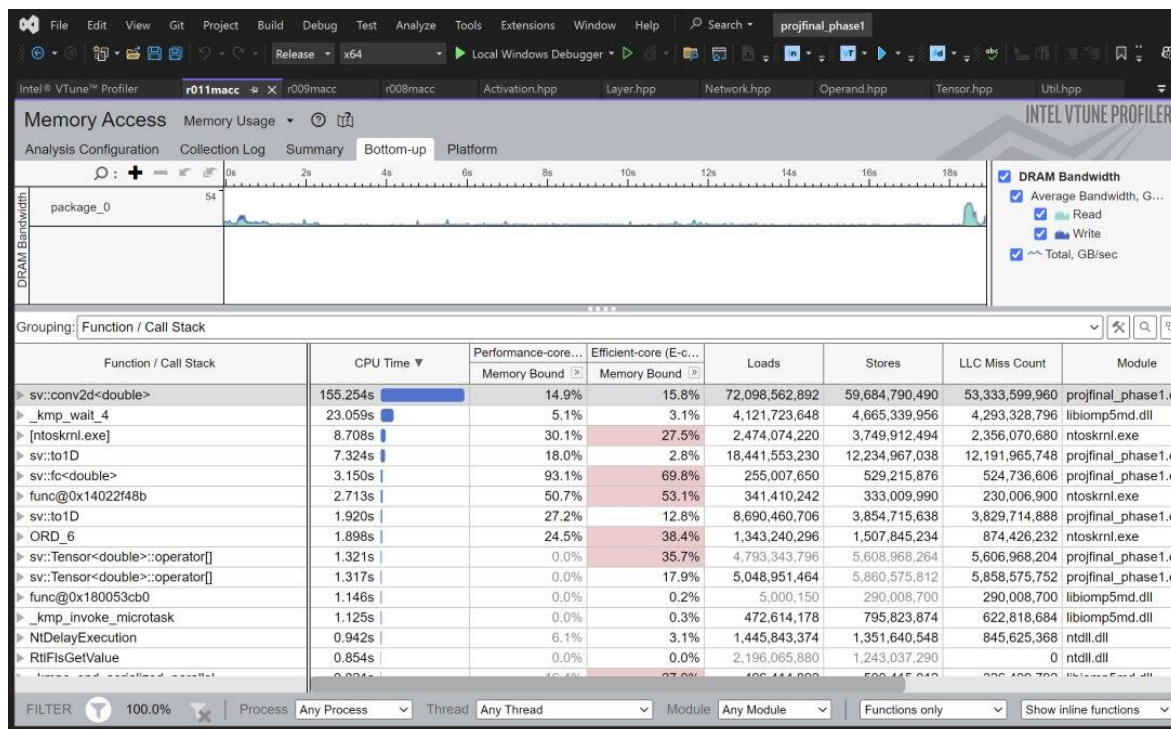
Hotspots Intel VTUNE PROFILER

Analysis Configuration Collection Log **Summary** Bottom-up Caller/Callee Top-down Tree Flame Graph Platform **Operand.hpp**

Source Assembly

Source Line	Source	CPU Time: Total	CPU Time: Self
53	#pragma omp parallel for collapse(3)	83.6%	25.170s
54	for (m = 0; m < w_f; m++) // kernel fmap		
55	{		
56	for (j = 0; j < w_h; j++) // kernel y		
57	{		
58	for (i = 0; i < w_w; i++) // kernel x		
59	{		
60	dtype inputWeight = input[sv::to1D(m, (y + j), (x + i), i	2.4%	4.977s
61	dtype kernelWeight = weight[sv::to1D(n, m, j, i, w_w, w_h	0.5%	1.140s
62	#pragma omp atomic		
63	sum += inputWeight * kernelWeight;	56.7%	118.704s
64	}		
65	}		
66	}		
67	#pragma omp critical		
68	{		
69	sum = blasfma		

■ آنالیز Memory Access



Memory Bound: این برنامه بیشتر به DRAM با معیار ۲۵.۶٪ محدود شده است. این نشان می‌دهد که عملکرد برنامه به طور قابل توجهی تحت تأثیر سرعت خواندن یا نوشتن داده‌ها از DRAM است.


پهنای باند DRAM: نمودار پهنای باند DRAM نوسانات را نشان می‌دهد. اگر استفاده از پهنای باند به طور مداوم بالا باشد، می‌تواند نشان دهد که برنامه واقعاً محدود به پهنای باند است.

LLC Miss Rate: تعداد خطاهای LLC برای توابع مختلف در یکی از ستون‌ها با مقادیر عددی در کنار هر فراخوانی تابع ذکر شده است. نرخ اشتباه LLC بالا می‌تواند منجر به افزایش تأخیر شود زیرا CPU منتظر می‌ماند تا داده‌ها از استخرهای حافظه کندتر fetch شوند. این می‌تواند بر عملکرد تأثیر منفی بگذارد، به‌ویژه اگر داده‌ها به ندرت استفاده شوند و دسترسی‌ها با از دست دادن تقاضا تداخل داشته باشند.

معیارهای عملکرد: معیارهای مختلفی، مانند میانگین پهنای باند، حداکثر پهنای باند، و تعداد خطاهای LLC، بینش‌هایی را در مورد میزان استفاده برنامه از منابع سیستم ارائه می‌دهد. به عنوان مثال، اگر پهنای باند متوسط نزدیک به حداکثر پهنای باند باشد، استفاده کارآمد از DRAM را پیشنهاد می‌کند. با این حال، اگر تعداد زیادی خطا از LLC وجود داشته باشد، می‌تواند به این معنی باشد که الگوهای دسترسی به داده‌های برنامه به خوبی برای استفاده از حافظه پنهان بهینه نشده است.






■ بهبود کد و تسريع


با توجه به نتايج حاصل از profiling پيشترين زمان مصرف شده در بخش operand.hpp و مربوط به خط زير مي باشد.

Hotspots 

Analysis ConfigurationCollection LogSummaryBottom-upCaller/CalleeTop-down TreeFlame GraphPlatformOperand.hpp x

SourceAssembly



Source Line ▲	Source	 CPU Time: Total ▾	CPU Time: Self ▾
53	<code>#pragma omp parallel for collapse(3)</code>	83.6%	25.170s
54	<code>for (m = 0; m < w_f; m++) // kernel fmap</code>		
55	<code>{</code>		
56	<code>for (j = 0; j < w_h; j++) // kernel y</code>		
57	<code>{</code>		
58	<code>for (i = 0; i < w_w; i++) // kernel x</code>		
59	<code>{</code>		
60	<code>dtype inputWeight = input[sv::to1D(m, (y + j), (x + i), i</code>	2.4%	4.977s
61	<code>dtype kernelWeight = weight[sv::to1D(n, m, j, i, w_w, w_h</code>	0.5%	1.140s
62	<code>#pragma omp atomic</code>		
63	<code>sum += inputWeight * kernelWeight;</code>	56.7%	118.704s
64	<code>}</code>		
65	<code>}</code>		
66	<code>}</code>		
67	<code>#pragma omp critical</code>		
68	<code>{</code>		
69	<code>sum += block_fmap</code>		

يکي از راه هايي که مي توان زمان صرف شده را کاهش داد unroll کردن حلقه for است.

باز کردن حلقه تکنیکی است که برای بهینه سازی حلقه ها با کاهش سربار کنترل حلقه و افزايش موازي بودن سطح دستورالعمل استفاده مي شود.

```
#pragma omp parallel for collapse(3)
for (n = 0; n < o_c; n++) // output channel
{
    for (y = 0; y < o_h; y++) // output y
    {
        for (x = 0; x < o_w; x++) // output x
        {
            dtype sum = 0;
            // Loop unrolling and blocking
            for (int block_y = 0; block_y < poolSize; block_y += 2)
            {
                for (int block_x = 0; block_x < poolSize; block_x += 2)
                {
                    for (m = 0; m < w_f; m++) // kernel fmap
                    {
                        for (j = block_y; j < std::min(block_y + 2, poolSize); j++)
                        {
                            for (i = block_x; i < std::min(block_x + 2, poolSize); i++)
                            {
                                dtype inputWeight = input[sv::to1D(m, (y + j), (x + i), i_w, i_h)];
                                dtype kernelWeight = weight[sv::to1D(n, m, j, i, w_w, w_h, w_f)];
                                sum += inputWeight * kernelWeight;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```

    }
    sum += bias[n];
    output[sv::to1D(n, y, x, o_w, o_h)] += sv::ReLU(sum);
  }
}
}

```

بیرونی‌ترین سه حلقه (`n` , y, x``) با استفاده از `"pragma omp#"` موازی OpenMP برای `collapse (۳)` موازی می‌شوند. این دستورالعمل به کامپایلر می‌گوید که تکرارهای این سه حلقه تودرتو را در چند رشته توزیع کند.

داخلی‌ترین حلقه‌ها با پردازش دو عنصر در یک زمان باز می‌شوند. این کار با افزایش شمارنده‌های حلقه (`"block_y"` و `"block_x"`) به جای یک عنصر انجام می‌شود.

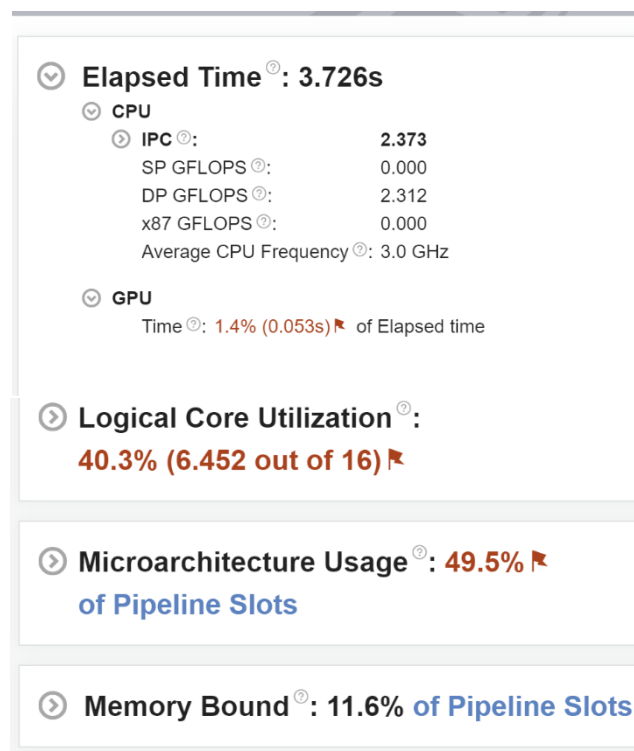
تابع `std::min` برای اطمینان از اینکه حلقه از مرزهای `poolSize` تجاوز نمی‌کند استفاده می‌شود. این ضروری است زیرا ما در یک زمان دو عنصر را پردازش می‌کنیم و باید بررسی کنیم که آیا به انتهای بلوک رسیده‌ایم یا خیر.

با باز کردن حلقه‌ها، تعداد دفعات اجرای کد کنترل حلقه را کاهش می‌دهیم. این می‌تواند منجر به استفاده بهتر از خط لوله دستورالعمل CPU و کاهش پیش‌بینی نادرست شاخه شود.

`blocking` نیز با تقسیم عملیات پیچیدگی به بلوک‌های کوچکتر که در حافظه پنهان CPU قرار می‌گیرند، اعمال می‌شود. این امر زمان دسترسی به حافظه را کاهش می‌دهد و می‌تواند عملکرد را به میزان قابل توجهی بهبود بخشد.

ترکیبی از باز کردن حلقه و `blocking` می‌تواند منجر به بهبود عملکرد قابل توجهی در عملیات کانولوشن شود، به ویژه هنگامی که با مجموعه داده‌های بزرگ یا تصاویر با وضوح بالا سروکار داریم. با این حال، توجه به این نکته مهم است که این بهینه‌سازی‌ها ممکن است پیچیدگی کد را افزایش داده و بر قابلیت نگهداری تأثیر بگذارد.

نتایج `profiling` پس از اعمال تغییرات:



اگر از ابزار hotspot برای پروفایل کردن استفاده کنیم، میزان بهبود سرعت کد مشاهده می‌شود.

ابتدا نتایج کد اصلی را مشاهده می‌کنیم.

Hotspots ⓘ ⓘ

Analysis ConfigurationCollection LogSummaryBottom-upCaller/CalleeTop-down Tree

⌵ Elapsed Time ⓘ: 19.429s

⌵ CPU Time ⓘ: 209.398s

Effective Time ⓘ: 168.036s

⌵ Spin Time ⓘ: 31.759s ⚡

⌵ Overhead Time ⓘ: 9.604s

Total Thread Count: 16

Paused Time ⓘ: 0s

⌵ Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time ⓘ	% of CPU Time ⓘ
sv::conv2d<double>	projfinal_phase1.exe	149.990s	71.6%
_kmpc_critical	libiomp5md.dll	34.299s ⚡	16.4% ⚡
sv::to1D	projfinal_phase1.exe	6.366s	3.0%
sv::fc<double>	projfinal_phase1.exe	3.957s	1.9%
func@0x180070650	libiomp5md.dll	2.183s	1.0%
[Others]	N/A*	12.603s	6.0%

*N/A is applied to non-summable metrics.

مشاهده می‌کنیم که تابع conv2D با ۱۴۹.۹ ثانیه زمان‌برترین تابع در کد است.

پس از اعمال تغییرات نتایج به صورت زیر می‌باشد.

Hotspots ⓘ ⓘ

Analysis ConfigurationCollection LogSummaryBottom-upCaller/CalleeTop-down Tree

⌵ CPU Time ⓘ: 40.375s

Instructions Retired: 155,302,280,000

⌵ Microarchitecture Usage ⓘ: N/A* of Pipeline Slots

Total Thread Count: 19

Paused Time ⓘ: 0s

⌵ Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time ⓘ	% of CPU Time ⓘ
sv::conv2d<double>	projfinal_phase1.exe	29.570s	73.2%
sv::fc<double>	projfinal_phase1.exe	3.164s	7.8%
func@0x140232497	ntoskrnl.exe	1.310s	3.2%
RtlFlsGetValue	ntdll.dll	1.032s	2.6%
func@0x1402f7fd0	ntoskrnl.exe	0.780s	1.9%
[Others]	N/A*	4.520s	11.2%

*N/A is applied to non-summable metrics.

این بار همین تابع زمان‌برترین تابع است اما با این تفاوت که زمان اجرای آن به ۲۹.۵۷ ثانیه کاهش یافته است.