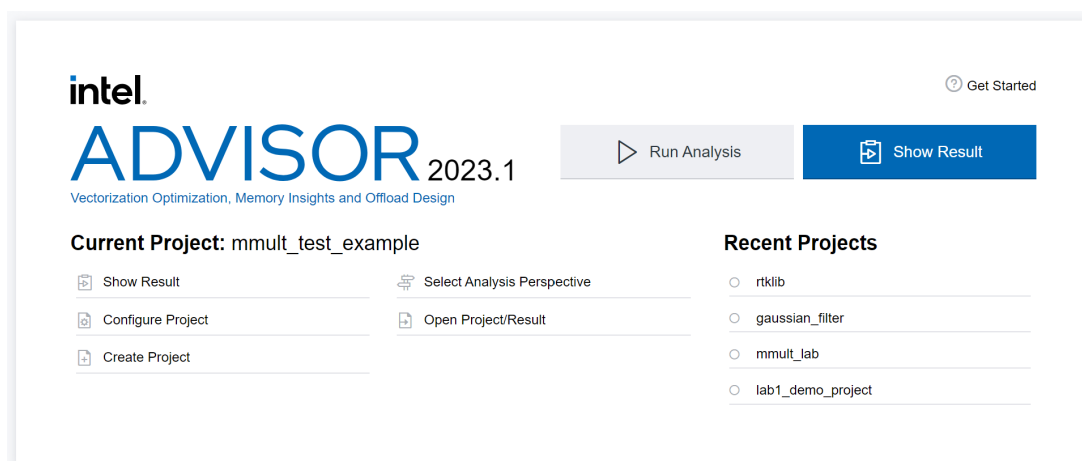


Lab2. Intel advisor

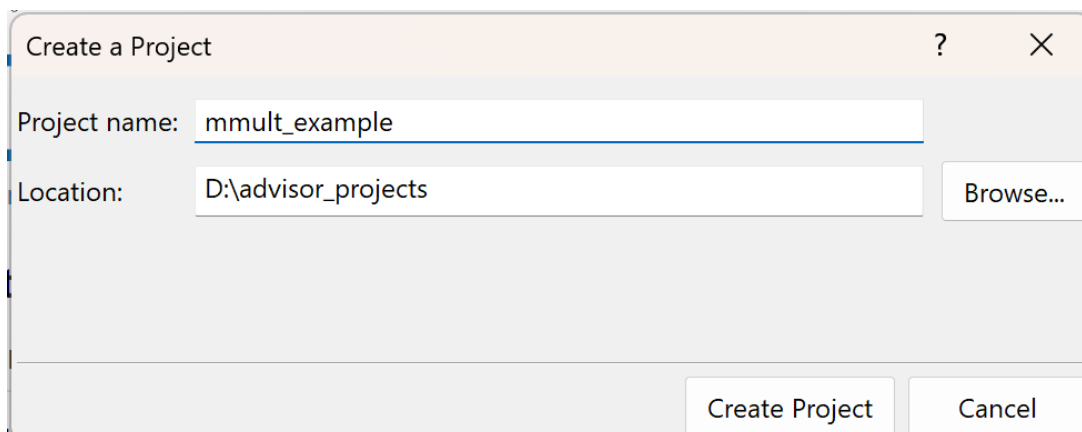
Этап с настройкой окружения для запуска Intel Advisor соответствует аналогичному шагу в предыдущей лабораторной работе. Запуск Intel Advisor с использованием пользовательского интерфейса (выполняется из консоли с настроенным окружением):

```
C:\Users\k.sandalov>advisor-gui
```

Для создания нового проекта - **Create project**:



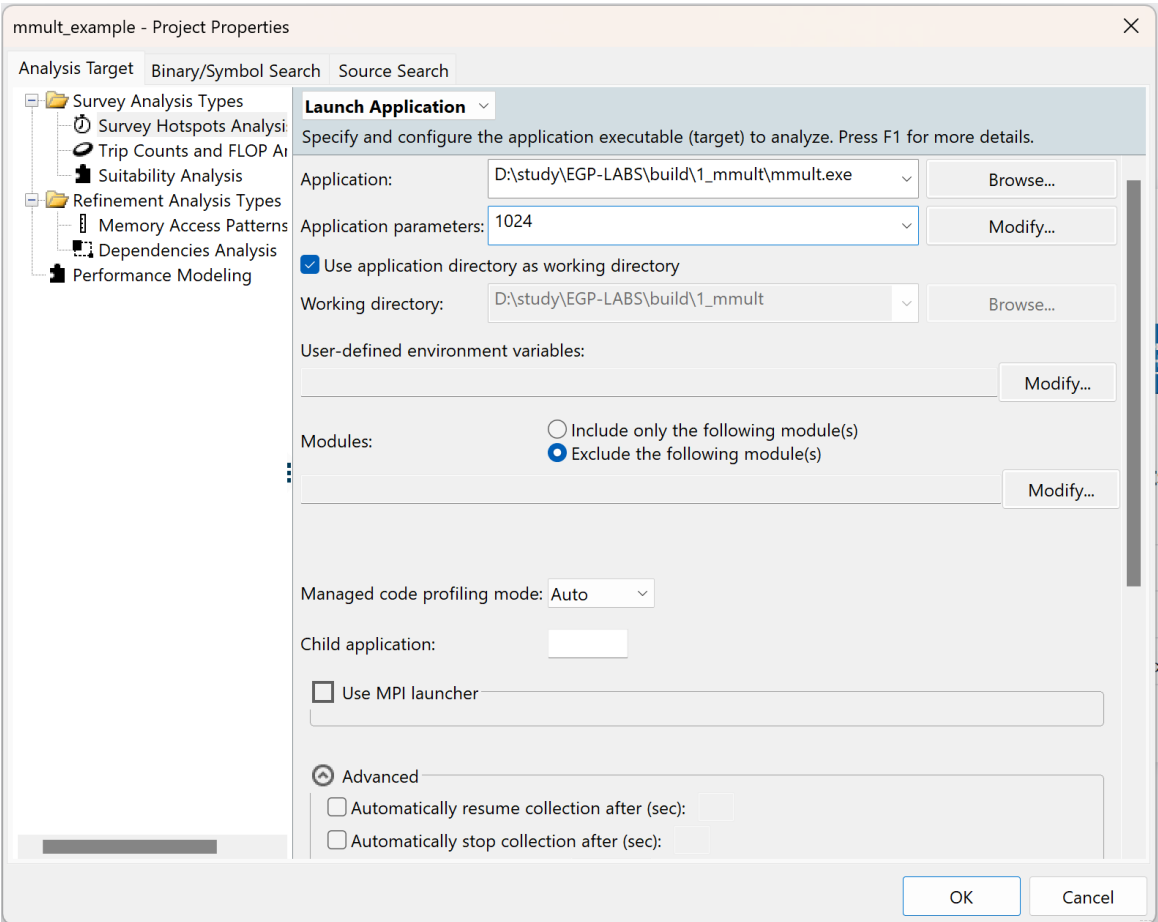
Выбор имени проекта и его расположение:



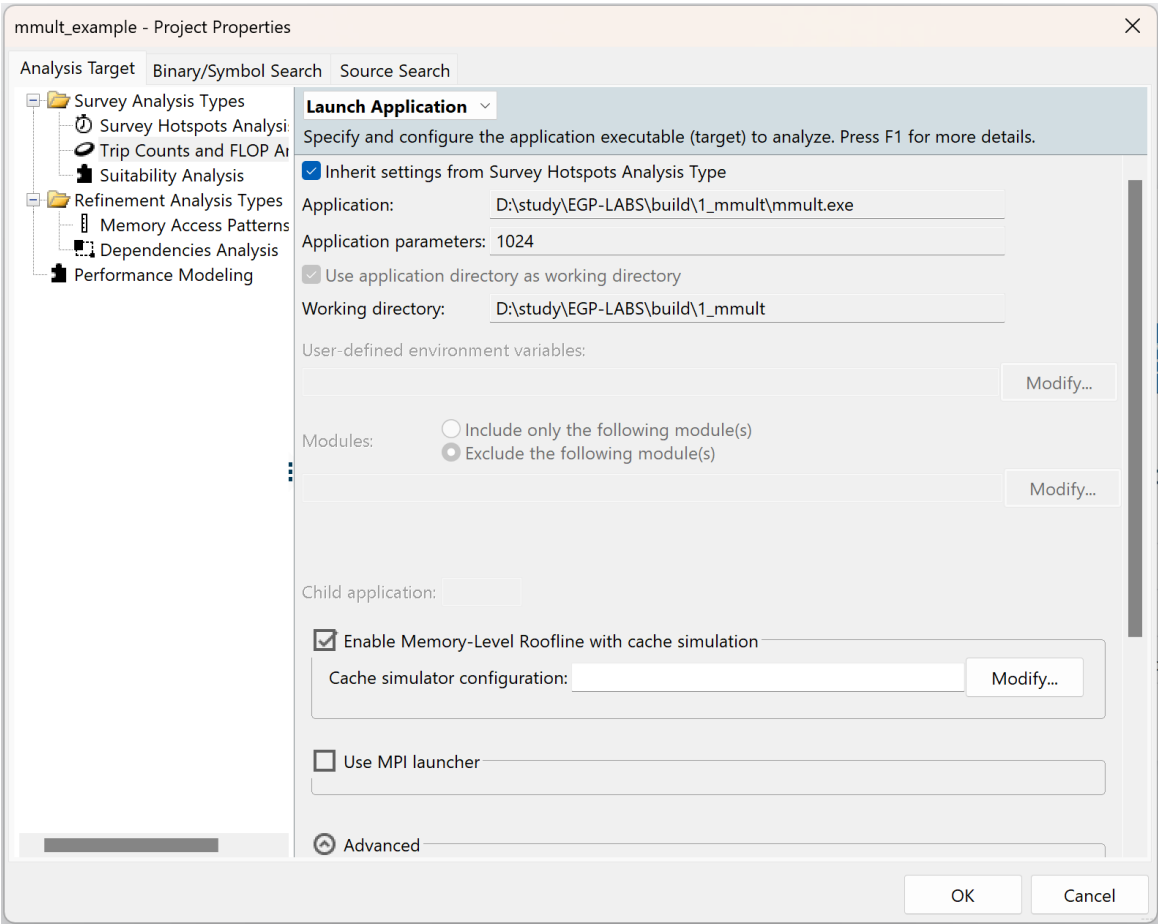
После этого, для начальной настройки проекта нужно на вкладке **Analysis Target** -> **Survey Hotspots Analysis** выбрать анализируемое приложение, указав полный путь до полученного .exe файла в окне **Application**. Соответственно, аргументы запускаемого приложения (выбран режим **Launch Application**), задаются в окне **Application parameters**.

В рамках лабораторной работы используем тестовое приложение mmult из этого репозитория. Сперва необходимо собрать его базовую версию со следующими опциями компиляции:

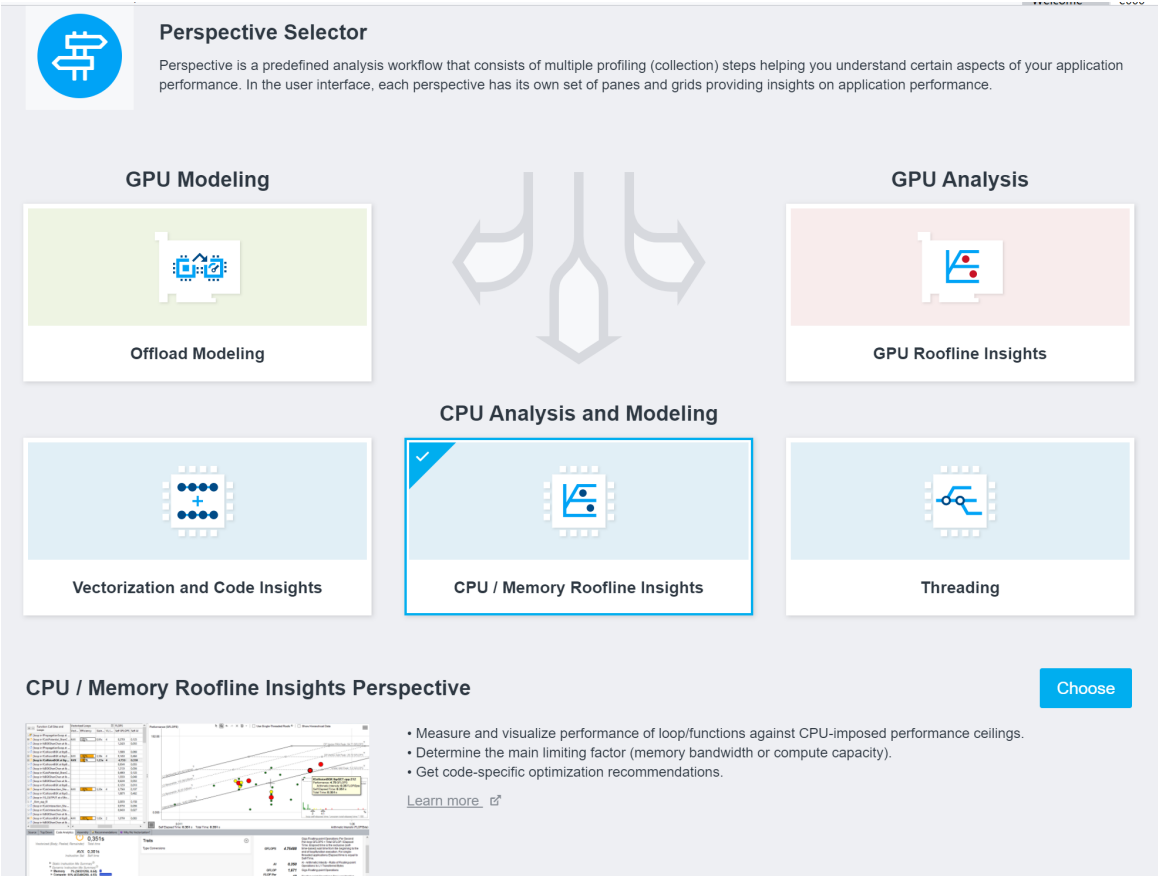
```
add_compile_options(-O3 -Qopt-report=max -debug)
```



Дополнительно на вкладке **Analysis Target** -> **Trip Counts and FLOP Analysis** необходимо выбрать пункт **Enable Memory-Level Roofline with cache simulation**:

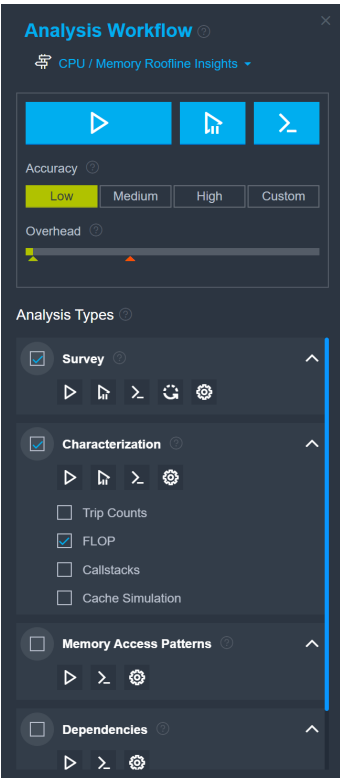


При создании нового проекта первым должно появиться окно с выбором определенного типа анализа или же, **Perspective selector**:



Нужно выбрать **CPU / Memory Roofline Insights**.

В блоке **Analysis Workflow** можно переключиться между разными перспективами а так же подобрать параметры для текущего типа анализа и запустить сбор данных + непосредственно сам анализ.



По типам анализов:

- **Survey** - начальное определение горячих участков приложение (hotspots).
- **Characterization** - более тяжеловесный анализ, позволяющий определить конкретнее, сколько операций с плавающей точкой было выполнено, сколько раз было вызвано тело анализируемого цикла, дерево вызовов соответствующих функций, замерить трафик уровня L1 и прочее. Для первого запуска должен быть выбран **Survey + Characterization (FLOP)**, сравнить время их выполнения.

Задание: Результаты сравнения добавить в отчет. Время выполнения анализа можно найти на вкладке **Summary** в блоке **Collection Details**:

▼ Collection Details

Survey

Collection started	21 February 2024, 03:06:16
Collection finished	21 February 2024, 03:06:17
Collection time	00 min 01 sec
Finalization time	00 min 01 sec
Full time	00 min 02 sec
Collection Log	See log
Application Output	See output
Collection Command Line	See command line

Summary:

На этой вкладке приведены общие сведения о производительности приложения:

- Количество выполняемых потоков
 - Время работы программы от старта выполнения первого потока до завершения последнего
 - Используемый набор векторных инструкций
 - Доля векторизованных вычислений
- $\text{GFLOPS} = \text{FLOPs} / \text{Seconds}$ (аналогично и INTOPS)
 - $\text{ArithmeticIntensity(AI)} = \text{FLOPs} / \text{Bytes}$ (отношение кол-ва вычислений к кол-ву запрошенных для этого байтов из памяти)

CPU / Memory Roofline Insights ▼

CPU / Memory Roofline Insights perspective measures and visualizes the actual performance of CPU kernels against hardware-imposed performance ceilings and determines the main limiting factor.

▼ Program Metrics

Program Elapsed Time	0,99s	▼ GFLOPS	2,16
Vector Instruction Set	SSE	GFLOP Count	2.000
Number of CPU Threads	1	FP Arithmetic Intensity ⓘ	<0.001
		▸ GINTOPS	0,55

▼ Performance Characteristics

Metrics	Total	
CPU Time	0,84s	<div><div></div></div> 100%
Time in scalar code	0,84s	<div><div></div></div> 100%

▼ Vectorization Gain/Efficiency (Not Available)

No vectorized loops found or not enough data

> OP/S And Bandwidth

▼ Per Program Recommendations

Higher instruction set architecture (ISA) available

Consider recompiling your application using a higher ISA. [Show more](#)

- Теоретические пиковые значения производительности для данной платформы и реально задействованная мощность:

OP/S And Bandwidth

Effective OP/S And Bandwidth		Utilization	Hardware Peak
GFLOPS	4.883	7.76%	out of 62.940 (DP) GFLOPS
		4.00%	out of 121.938 (SP) GFLOPS
GINTOPS	1.247	2.88%	out of 43.286 (Int64) GINTOPS
		1.31%	out of 95.035 (Int32) GINTOPS
CPU <-> Memory [L1+NTS GB/s]	29.326	7.43%	out of 394.856 GB/s [bytes]

Тут же можно найти рекомендации, применимые глобально ко всему приложению, например: Использовать более широкий набор инструкций.

Задание: Зафиксировать время выполнения программы, факт использования векторных инструкций, текущие GFLOPS и AI для приложения. Пиковые значения пропускной способности вычислительной системы.

Топ самых тяжелых хотспотов и рекомендации для них. Подробнее про анализ хотспотов на вкладке **Survey and Roofline**:

Per Program Recommendations

Higher instruction set architecture (ISA) available
Consider recompiling your application using a higher ISA. [Show more](#)

Top Time-Consuming Loops

Loop	Self Time	Total Time
loop in multiply_simple<float> at mmult.cpp:45	0.820s	0.820s
loop in operator new at new_scalar.cpp:35	<0.001s	0.003s
loop in multiply_simple<float> at mmult.cpp:40	<0.001s	0.820s
loop in multiply_simple<float> at mmult.cpp:42	<0.001s	0.820s
loop in mmult_task at mmult.cpp:131	<0.001s	0.003s

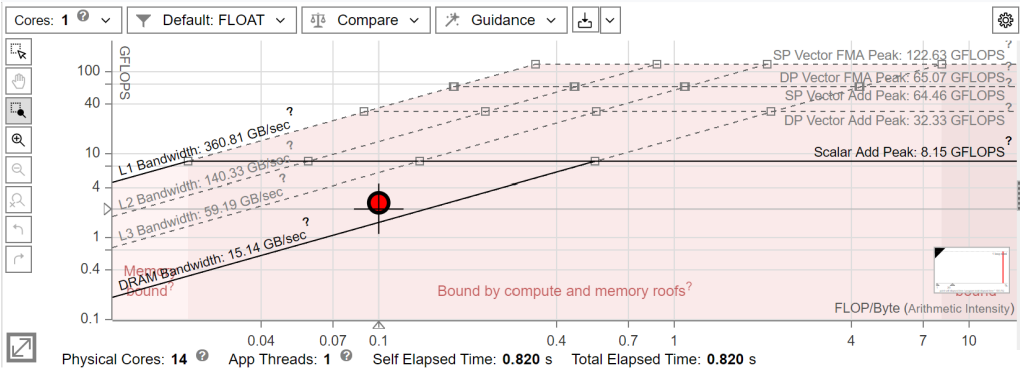
Suitability And Dependencies Analysis Data

No data available. Collect Suitability or Dependency to see the results.

Recommendations

- Target the AVX2 ISA loop in multiply_simple<float> at mmult.cpp:45
- Remove system function call(s) inside loop loop in operator new at new_scalar.cpp:35
- Remove system function call(s) inside loop loop in mmult_task at mmult.cpp:200

Roofline:



На графике руфлайна наглядно отображены ограничения пропускной способности (вычислительной и подсистемы памяти), а также то, насколько эффективно их использует тестовое приложение. На графике отображаются точки, характеризующие производительность самых горячих участков тестового

приложения. И в данном случае, это цикл на строке 45 в mmult.cpp. Для этой точки вычисляются значения GFLOPS/GINTOPS и AI, а так же, исходя из используемого набора инструкций, типа операций (FP или INT) и конфигурации подсистемы памяти ограничивающие ее крыши.

Соответственно, ограничением производительности будет служить $\text{MIN}(\text{peakMemBandwidth} \times \text{AI}, \text{peakGFLOPS})$

Survey:

Function Call Sites and Loops	Performance Issues	CPU Time		Type	Why No Ve
		Total Time	Self Time		
[loop in multiply_simple<float> at mmult.cpp:45]	1 Potential un...	0,820s	0,820s	Scalar	
[loop in operator new at new_scalar.cpp:35]	1 System functi...	0,003s	0,000s	Scalar	
mainCRTStartup		0,839s	0,000s	Function	
__srt_common_main		0,839s	0,000s	Function	
__srt_common_main_seh		0,839s	0,000s	Function	
invoke_main		0,839s	0,000s	Function	
main		0,839s	0,000s	Function	
[loop in mmult_task at mmult.cpp:131]		0,003s	0,000s	Scalar	
mmult_task		0,839s	0,000s	Function	
operator new[]		0,003s	0,000s	Function	
operator new		0,003s	0,000s	Function	
multiply_simple<float>		0,820s	97,8%	Function	

Bottom-up представление списка хотспотов приложения в виде таблицы с performance метриками.

Задание: Определить главный хотспот. Затем выделить ограничивающие его крыши, зафиксировать в отчете.

Ниже вкладки:

Source:

Source Top Down Code Analytics Assembly Recommendations Why No Vectorization?					
File: mmult.cpp:45 multiply_simple<float>					
Li.	Source	Total Time	%	Loop/Function Time	Traits
37	template <typename val_t>				
38	void multiply_simple(int arrSize, val_t **aMatrix, val_t **bMatrix, val_t **cMatrix)				
39	{				
40	for (int i = 0; i < arrSize; i++)				
41	{				
42	for (int j = 0; j < arrSize; j++)				
43	{				
44	#pragma novector				
45	for (int k = 0; k < arrSize; k++)	75,570ms		820,096ms	
	[loop in multiply_simple<float> at mmult.cpp:45]				
	Scalar loop				
Selected (Total Time):		75,570ms			

Выделено непосредственно место в исходном коде, соответствующее выбранному хотспоту.

Top Down

Source		Top Down		Code Analytics		Assembly		💡 Recommendations		🚫 Why No Vectorization?	
Function Call Sites and Loops						CPU Time			Type	Why	
						Total Time %	Total Time	Self Time			
[-] _srt_common_main						100,0%	<div><div></div></div>	0,839s	0,000s	Function	
[-] _srt_common_main_seh						100,0%	<div><div></div></div>	0,839s	0,000s	Function	
[-] invoke_main						100,0%	<div><div></div></div>	0,839s	0,000s	Function	
[-] main						100,0%	<div><div></div></div>	0,839s	0,000s	Function	
[-] mmult_task						100,0%	<div><div></div></div>	0,839s	0,000s	Function	
[-] multiply_simple<float>						97,8%	<div><div></div></div>	0,820s	0,000s	Function	
[-] [loop in multiply_simple<float> at mmult.cpp:40]						97,8%	<div><div></div></div>	0,820s	0,000s	Scalar	
[-] [loop in multiply_simple<float> at mmult.cpp:42]						97,8%	<div><div></div></div>	0,820s	0,000s	Scalar	
[-] [loop in multiply_simple<float> at mmult.cpp:45]						97,8%	<div><div></div></div>	0,820s	0,820s	Scalar	
[+] [loop in mmult_task at mmult.cpp:200]						1,8%	<div><div></div></div>	0,015s	0,000s	Scalar	
[+] [loop in mmult_task at mmult.cpp:131]						0,4%	<div><div></div></div>	0,003s	0,000s	Scalar	

Дерево вызовов, для выбранного хотспота. Соответственно, для одного хотспота может быть выделено сразу несколько стеков вызовов. И тут уже можно посмотреть долю каждого в суммарном времени для хотспота.

Code Analytics - суммарная информация по хостпоту + по миксу инструкций для него (соотношение разных типов инструкций - memory vs compute)

Recommendations

SourceTop DownCode AnalyticsAssemblyRecommendationsWhy No Vectorization?

! Potential underutilization of FMA instructions

Your current hardware supports the AVX2 instruction set architecture (ISA), which enables the use of fused multiply-add (FMA) instructions. Improve performance by utilizing FMA instructions.

Target the higher ISA

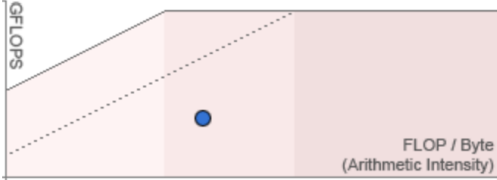
Although static analysis presumes the loop may benefit from FMA instructions available with the AVX2 or higher ISA, no FMA instructions executed for this loop. To fix: Use the following compiler options:

- xCORE-AVX2 to compile for machines with and without AVX2 support
- axCORE-AVX2 to compile for machines with AVX2 support only
- xCOMMON-AVX512 to compile for machines with AVX-512 support only
- axCOMMON-AVX512 to compile for machines with and without AVX-512 support

Note: the compiler options may vary depending on the CPU microarchitecture.

! Roofline conclusions

Conclusions, with optimization recommendations, are sorted by relevance.



This loop is mostly memory bound but may also be compute bound

The performance of the loop is bounded by the bandwidth of the shared cache and DRAM. To improve performance: Improve caching efficiency. The loop is also scalar. To fix: Vectorize the loop.

Collect Roofline for all memory levels

Run the Roofline for all memory levels to get a detailed analysis of memory-bound loops/functions.

Memory-Level Roofline evaluates the traffic between each memory subsystem based on cache simulation data.

Potential underutilization of FMA instructions

Target the higher ISA

Roofline conclusions

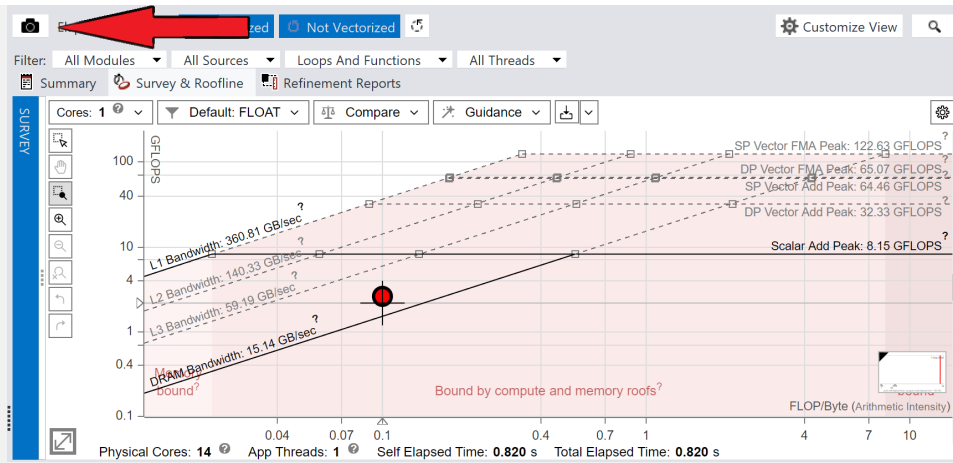
This loop is mostly memory bound but may also be compute bound

Collect Roofline for all memory levels

То, на что стоит обратить большее внимание. Довольно важной частью данного инструмента является система предоставления рекомендаций по анализируемому коду, советы, на что стоит обратить внимание, и даже, возможно, как можно исправить те или иные проблемы с производительностью приложения.

Далее перед любыми модификациями приложения необходимо делать снимки:

7 / 11



Create a Result Snapshot

Result name:

☒ Cache sources ☒ Cache binaries

☒ Pack into archive

Result path:

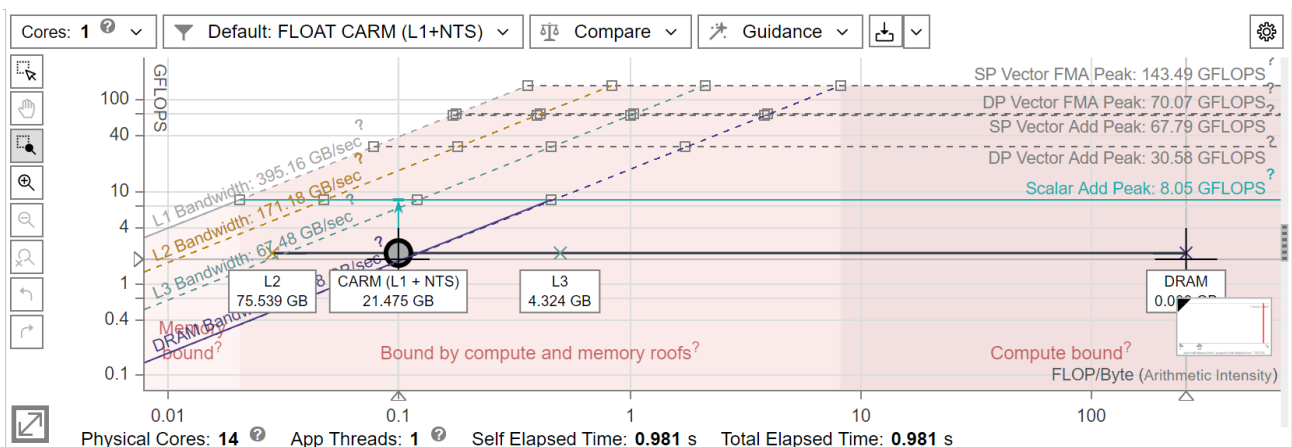
Снимки профиля, которые далее можно будет сравнивать между собой.

Итак, в данном примере нам предлагается две рекомендации:

- Использовать более широкий набор инструкций + FMA
- Собрать Roofline для всех уровней памяти.

Первое выполнять пока нет смысла, так как в самом коде у нас явно отключена векторизация. Тогда попробуем собрать Memory Level Roofline.

Для этого необходимо в панели **Analysis Workflow**, в **Characterization** блоке выбрать пункт **Cache Simulation** и пересобрать отчет.



И теперь по двойному клику на точку на графике руфлайна раскроется раскладка для данного хотспота по утилизации различных уровней подсистемы памяти.

! Possible inefficient memory access patterns present

Inefficient memory access patterns may result in significant vector code execution slowdown or block automatic vectorization by the compiler. Improve performance by investigating.

Confirm inefficient memory access patterns

There is no confirmation inefficient memory access patterns are present. To fix: Run a [Memory Access Patterns analysis](#).

А в рекомендациях появится новая опция-предложение: запустить MAP анализ, который позволит определить паттерны доступа к памяти в рамках хотспота.

Для этого, во-первых, необходимо выделить интересующие нас хотспоты:

Function Call Sites and Loops	Performance Issues	CPU Time		Type	Why No Vectoriz
		Total Time	Self Time		
[loop in multiply_simple<float> at mmult.cpp:45]	2 Mismatched lo...	0,105s	0,105s	Inside vectorized	
mainCRTStartup		0,105s	0,000s	Function	
__srt_common_main		0,105s	0,000s	Function	
__srt_common_main_seh		0,105s	0,000s	Function	
invoke_main		0,105s	0,000s	Function	
main		0,105s	0,000s	Function	
mmult_task		0,105s	0,000s	Function	
multiply_simple<float>		0,105s	0,000s	Function	
[loop in multiply_simple<float> at mmult.cpp:40]		0,105s	0,000s	Vectorized (Body)	
[loop in multiply_simple<float> at mmult.cpp:40]		0,105s	0,000s	Scalar	

И выбрать в панели **Analysis Workflow** новый анализ - **Memory Access Patterns** и запустить его.

Третья вкладка **Refinement Reports**:

Тут можно найти отчет по проведенному MAP анализу:

Site Location	Loop-Carried Dependencies	Strides Distribution	Access Pattern	Footprint Estimate Max. Per-Instruction Addr. R
[loop in multiply_simple<float> at mmult.cpp:40]	No Information Available	50% / 0% / 50%	Mixed Strides	448MB
[loop in multiply_simple<float> at mmult.cpp:42]	No Information Available	100% / 0% / 0%	All Unit Strides	447MB
[loop in multiply_simple<float> at mmult.cpp:45]	No Information Available	33% / 33% / 33%	Mixed Strides	447MB
<pre>43 { 44 #pragma vector 45 for (int k = 0; k < arrSize; k++) 46 { 47 product[i][j] += aMatrix[i][k] * bMatrix[k][j]; </pre>				

Где, собственно, и написано, что в нашей программе, в хотспоте, обнаружен неэффективный паттерн доступа к памяти. А ниже приведены рокомендации, как от этого можно избавиться:

Memory Access Patterns Report

Dependencies Report

Recommendations

All Advisor-detectable issues: [C++](#) | [Fortran](#)

!

Inefficient memory access patterns present

There is a high of percentage memory instructions with irregular (variable or random) stride accesses. Improve performance by investigating and handling accordingly.

💡

Reorder loops

This loop has less efficient memory access patterns than a nearby outer loop. To fix: Reorder the loops if possible.

Example (original code)

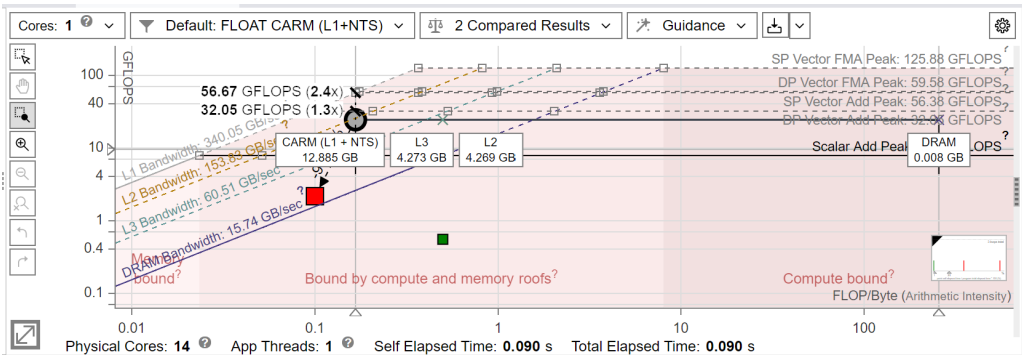
```
...
for (int j = 0; j < N; j++)
  for (int k = 0; k < N; k++)
    c[i][j] = c[i][j] + a[i][k] * b[k][j];
...
```

Example (revised code)

```
...
for (int k = 0; k < N; k++)
  for (int j = 0; j < N; j++)
    c[i][j] = c[i][j] + a[i][k] * b[k][j];
...
```

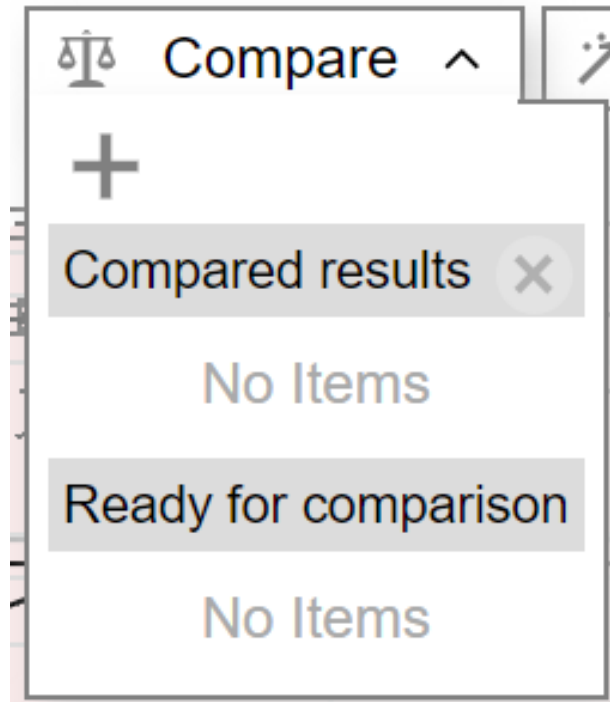
Сохраняем снимок и применяем исправления в коде. Снимок приложить к отчету. Можно собрать все в архив и сохранить для дальнейшей отправки преподавателю.

Memory Level Roofline для новой версии приложения:

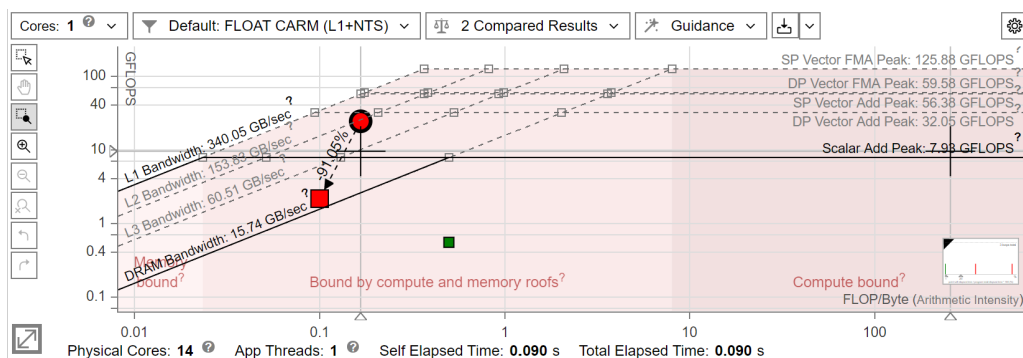


Доп. вопрос: о чем на рисунке выше говорит подобное расположение точек для разных уровней памяти на графике руффлайна.

Теперь можно сравнить две версии отчета соотв. разным версиям приложения:



Добавляем сохраненный ранее снимок и получаем следующую картинку:



Наглядно видно прирост производительности.

Задание: добавить векторизацию `#pragma novector` --> `#pragma vector`. Сравнить результат выполнения векторизованного кода с не векторизованным. Использовать AVX

```
add_compile_options(-O3 -Qopt-report=max -debug /QxCORE-AVX2 /Qalign-loops:32)
```