インテル[®] VTune[™] Profiler

エクセルソフト株式会社



インテル[®] VTune[™] Profiler

インテル® プロセッサーで動作するプログラムを最適化するためのプロファイラー

CPU の使用率や、メモリーの利用状況を数 値化、グラフ化して表示

パフォーマンス問題のある個所を自動的に 検出

修正の効果があると見込めるところのみを 表示

Advanced Hotspots Hotspots + 0 INTEL VTUNE AMPLIFIER 2019										
Analysis Configuration Collect	tion Log Summary E	Bottom-up Calle	er/Callee	Top-down	Tree Plat	form				
Grouping: Function / Call Stack										
		CPU Time 🔻		K	Context S	witch Time 📧	Context Switt ^	1		
Function / Call Stack	Effective Time by U Idle Poor Ok	Jtilization 🔉 🔊	Spin Time	Overhead Time	Wait Time	Inactive Time	Preemption	J		
▼ updateBusinessAccount	7.915s		0s	0s	0s	0.055s	934	l		
main\$omp\$parallel_for@269	7.915s		0s	0s	0s	0.055s	934	ŝ		
Kmp_invoke_microtas	7.915s <mark>–</mark>		0s	0s	0s	0.042s	815	i		
updateBusinessAccoun	t Os		0s	0s	0s	0.013s	119	l		
updateCustomerAccount	7.766s		0s	0s	0s	0.052s	1,111	l		
kmpc_atomic_fixed8_add	2.772s 📔		Os	0s				l		
kmpc_critical	Os		2.021s	0s	0s	0.014s	262 v	l		
< >	<						>	l		
p: 🕇 🗕 🖝 🖝	5s 5.2s 5	.4s 5.6s	5.8s	6s 6.	2s	✓ Thread	~ ^	١		
B OMP Worker Thread #2 (TI	and during the state of the	بالبيها بواري		الملامد	11.1.1.^	Runr	ing	l		
E OMP Worker Thread #3 (TI				in in the second second		Contex	Switches			
	تداعة الطنديل تلبيناك ستصنياتهم	و الزار، مامال اعتبار مرعاد الارار 	lan di kana da kata na sa	با البين القصاه	<u>ام ا</u> لعربان	Pre	emption			
rtmtest_openmp (TID: 12732)	والجافية بتبقح هعربوس أصركية	عالمعف ومسيبة مليام	ليبلغ تواسية إحرابه	بديناؤه هيبواب	ملته، بها	Syr	chronization			
OMP Worker Thread #1 (TI	وراهيان بقرابة وترجلت فراجأ فراه وبالأرفاء	بالمنابعة والتوارين	له سوطنها	الاستقرر الرقا	يويلين	CPU 📥	Time	1		
CPU Time	المالا ومعادية محمدة والمعادية				يو و و و ا	🗹 🖦 Spin	and Overh			
	<				>	CPU	_CLK_UNH 🗸	/		
FILTER 🝸 100.0% 🦕	Any Proc ~ Any Thr	read 🗸 Any M	Moc ~ An	yt~ [Jser functi ~	Show inl ~	Functior ~			





どこに時間を費やしているか

関数ごとの時間や呼び出し経路

Grouping:	Function / Call Stack		
Fund	tion / Call Stack	CPU Time 🔻 🔺	Source File
	ersect	3.844s	grid.cpp
sphere_	intersect	2.780s	sphere.cpp
Intersect ■ grid_bounds_intersect		0.291s	grid.cpp
shader		0.162s	shade.cpp

長い待機があるか

待機した時間と理由、スレッドの並行動作状況

Wait Time by Thread Concurrency ▼	Wait Count	Spin Time
4.036s	10	0s
4.014s	288	0.016s
0.714s	49	0s
0.649s	2	0.016s

効率の^亜い実行をしているか キャッシュミスや頻繁なメモリーアクセスなどイベントとソースコードの対応

		Back-End Bo					
Source		Memory Bound					
		DRAM Bound L3 Bound		ł			
		LLC Miss	Conte	Data	LLC Hit		
<pre>for(int k=0; k<n; k++)<="" pre=""></n;></pre>	11.385	0.000	0.000	0.000	0.000		
<pre>for(int j=0; j<p; j++)<="" pre=""></p;></pre>	3.618	0.000	0.000	0.000	0.000		
c[i][j] = c[i][j] + a[i][k] * b[k][j];	4.436	0.342	0.014	0.094	0.261		



バージョン 2019 の新機能 インテル[®] VTune[™] Profiler

インテル[®] Optane[™] DC パーシステント・メモリーと 第2世代インテル[®] Xeon[®] スケーラブル・プロセッサーの サポートを追加

プラットフォーム・プロファイラーの追加

- ハードウェア構成の問題点を検出
- 適切にチューニングされていない
 アプリケーションを特定

MPI プリケーションの解析を改善

• MPI_PControl API によるデータ収集をサポート





さまざまなシナリオに対応できるプロファイル機能





プログラム言語、ターゲットのサポート

C/C++、Fortran、.NET*、Java*、Python*、Go* ・ネイティブコード、マネージドコードに対応

CPUと内蔵 GPUの両方を使用したヘテロジニアス構成

• OpenCL* と C/C++ など

コンテナー上で実行されるC/C++、Fortran および、 Java* で記述されたアプリケーション (Linux* ホスト)

• LXC, Docker*, Mesos*, Singularity*



提供される解析タイプ

解析タイプは目的に応じて選択します



出典: https://software.intel.com/en-us/VTune-Profiler-help-analysis-ty



インテル[®] VTune[™] Profiler 画面構成

Navigator	🗄 🕂 🕨 📩 🜗 🗁 🕐 Welcome	x r004lw x r002hs x r003hs x	r001tr ×	r006gh ×	r000tr ×		=					
าล	Locks and Waits Locks and Wa	Cocks and Waits Locks and Waits • ⑦										
	Analysis Configuration Collection Log Summary Bottom-up Caller/Callee Top-down Tree Platform											
ble	Grouping: Sync Object / Function / Call Stat	k					~ 🛠 D 🖫					
ort yon	Sync Object / Function / Call Stack	Wait Time by Thread Concurrency ▼ ■ Idle ■ Poor ■ Ok ■ Ideal ■ Over	Wait Count	Spin Time	Module	Object Type	Object Creation					
Dtr	Multiple Objects	238.435s 📕					[Unknown]![Unknown					
tr	Critical Section 0xfbab6eac	147.789s	509	2.473s		Critical Section	analyze_locks.exe!ptl					
hs	Manual Reset Event 0xce58f39e	92.101s 	1	0s		Manual Reset Ever	nt gdiplus.dll!GdiplusSta					
าร	Auto Reset Event 0x08c67705	51.941s 📒	1,337	0s		Auto Reset Event	analyze_locks.exe!vic					
lw	Auto Reset Event 0xe6212da0	30.163s 🛑	3	0.009s		Auto Reset Event	tbb.dll!tbb::internal::g					
icgc	▶ Sleep	9.733s 🥚	915	0.181s		Constant	[Unknown]![Unknown					
ògh	TBB Scheduler	0.320s	4	0.070s		Constant	tbb.dll!tbb::internal::g					
	Thread 0x9ad15d80	0.053s	1	0.006s		Thread	gdiplus.dll!GdiplusSta					
	Stream\dat\balls.dat 0x24838b26	0.029s	128	0s		Stream	ucrtbase.dll!fopen					
	IME Msgs	0.012s	9	0s		Constant	[Unknown]![Unknown					
	Thread 0xee8b08fd	0.011s	1	0s		Thread	analyze_locks.exe!vic					
	Message	0.003s	6	0.008s		Message	[Unknown]![Unknown					
	Completion Port 0xf51f9a3f	0.002s	2	0s		Completion Port	ntdll.dll!KiUserCallbac					
	Unknown 0xbaa589fb	0.001s	1	0s		Unknown	CoreMessaging.dll!fu					
	Stream C:\WINDOWS\Globalization\Sorti	0.000s	1	0s		Stream	user32.dll!func@0x6					
	Read/Write Lock 0x3df0c762	0.000s	3	0s		Read/Write Lock	user32.dll!PeekMessi v					
	< ,	٤ 					>					
	Q:	20s 40s 60s	80	s 	100s	120s	🛛 Thread 🗸 🗸					
	WinMainCRTStartup (TID: 3					^	Running					
	threadstartex (TID: 13288)	(alight)	استاب المراجع	here we will be all	ulili sono a		✓ Waits					
	TBB Worker Thread (TID: 1			(with writed		CPU Time					
	thread video (TID: 1696)				a serie a serie a		Spin and Overhead					
	tillead_video (Hb. 1050)				and the second second		☐ ♥ CPU Sample					
	TBB Worker Thread (TID: 832)		didelate the second									
	func@0x100734a0 (TID: 13						CPU Utilization					
	func@0x10068430 (TID: 872)						CPU Time					
	func@0x10057920 (TID: 11						Spin and Overhead					
	func@0x10068430 (TID: 12					v						
	CPI I Utilization											

| メニューバーと | タブ

解析結果の表示



プロジェクト・ ナビゲーター

※スタンドアロンのみ

パフォーマンス向上の可能性を表示

性能問題となりえそうな項目を赤く表示します

・注目すべき個所を指摘

🌌 Threading Threading Efficiency 👻 🕐							
Analysis Configuration Collection Log Summary Bottom-up Caller/Callee Top-down Tree Platform							
Selapsed Time ^⑦ : 43.161s Paused Time ^⑦ : 0s							
 Effective CPU Utilization ⁽³⁾: 43.3% (1.732 out of 4 logical CPUs) ⁽³⁾ Effective CPU Utilization Histogram OpenMP Analysis. Collection Time ⁽³⁾: 43.161 Serial Time (outside parallel regions) ⁽³⁾: 4.885s (11.3%) Parallel Region Time ⁽³⁾: 38.276s (88.7%) Estimated Ideal Time ⁽³⁾: 26.329 (60.1%) OpenMP Potential Gain ⁽³⁾: 12.344s (28.6%) ⁽¹⁾ Top OpenMP Regions by Potential Gain 							
\odot Total Thread Count: 4 C Thread Oversubscription $^{\odot}$: 0s (0.0% of CPU Time)							
 Wait Time with poor CPU Utilization: 46.850s (100.0% of Wait Time) Top Waiting Objects 							
\odot Spin and Overhead Time $^{\odot}$: 1.003s (1.3% of CPU Time)							
S Collection and Platform Info							



注目すべき処理のソースコードを確認する

対象のオブジェクトをダブルクリックすることで、 ソースコード・レベルに情報を掘り下げます

Grouping: Function / Call Stack								
Function / Call Stack	Serial CPU Time	Effective						
multiply3\$omp\$parallel_for@	0s	38.334s						
omp_get_max_threads	2.214s	0s						
▶ allrem	0s	0.360s						
kmp_fork_barrier	0s	0s						
▶ alldiv	0s	0.194s						
▶kmp_join_call	0s	0s						
▶ init_arr	0.090s	0.090s						
▶ free	0.054s	0.054s						
▶ exit	0.014s	0.014s						
[Unknown]	00							

Sou	urce Assembly III = 👫 😽 😽 🍇							
		👍 CPU Time: Total						
SOL 🛦	Source	Effective Time by Utilization	Spin Time					
202	}							
203	void multiply3(int msize, int tidx, int numt, TYPE a[][N							
204	(
205	int i,j,k;							
206								
207	<pre>#pragma omp parallel for collapse (2)</pre>	91.9%	0.09					
208	for(i=0; i <msize; i++)="" td="" {<=""><td>0.6%</td><td>0.09</td></msize;>	0.6%	0.09					
209	for(k=0; k <msize; k++)="" td="" {<=""><td>0.1%</td><td>0.09</td></msize;>	0.1%	0.09					
210	#pragma ivdep							
211	for(j=0; j <msize; j++)="" td="" {<=""><td>2.6% 📒</td><td>0.09</td></msize;>	2.6% 📒	0.09					
212	c[i][j] = c[i][j] + a[i][k] * b[k][j];	88.6%	0.09					
213	}							
214	}							
215								



Hotspots 解析結果例

Bottom-up (関数/項目ごとの表示)

Basic Hotspots Hotspots by CPU Usage viewpoint (change)									
🔹 🐵 Analysis Target 🔥 Analysis Type 🔛 Collection Log 🛍 Summary 😪 Bottom-up 🚱 Caller/Callee 🚱 Top-down Tree 🔜 Platform									
Grouping: Function / Call Stack 🗸 🔖 🔍 🛠									
	CPU Time+		~			/			
Function / Call Stack	Effective Time by Utilization	Spin	Ove	Module	Function (Full)	Source File			
	🔲 Idle 📕 Poor 📙 Ok 📕 Ideal 📕 Over	Time	Time						
initialize_2D_buffer	9.921s	0s	0s	find_hotspots.exe	initialize_2D_buffer(unsigne	find_hotspots.cpp			
⊞ grid_intersect	3.751s	0s	0s	find_hotspots.exe	grid_intersect	grid.cpp			
sphere_intersect	1.922s	0s	0s	find_hotspots.exe	sphere_intersect	sphere.cpp			
	0.688s	0s	0s	USER32.dll	DispatchMessageA				
	0.422s	0s	0s	gdiplus.dll	GdipDrawImagePointRectI				
∃grid_bounds+	$-\mathcal{T} \Box \mathcal{T} \exists \mathcal{L}$ (find botcoots o		ካጣዞ	月米力 pots.exe	grid_bounds_intersect	grid.cpp			
⊞ Raypnt		xe) i ×± □	ADJA	司女X pots.exe	Raypnt(struct ray *,double)	vector.cpp			
⊞ shader	Initialize_2D_buffer について	有日		pots.exe	shader(struct ray *)	shade.cpp			
⊞libm_sse2_sqrt_precise	0.094s	0s	0s	ucrtbase.dll	libm_sse2_sqrt_precise				



Hotspots 解析結果例

4	🕽 Analysis Target 🔼 Analysis Type 🧱 Collection Log 🛍 Summary 🍫 Bottom-up 🍣 Caller/	Callee 😽 Top-down Tree 🔣 Platform	b find_hots	s 💥 🕞 🖓
So	irce Assembly 🗉 🗄 💿 💀 🧐 🚱 🗣 🔍 Assembly grouping: Address		~	CPU Time 🗸
		CPU Time: Total	^	Viewing ↓ 1 of 1 ▷ selected stack(s)
So.	Source	Effective Time by Utilization	»	100.0% (9.921s of 9.921s)
- F		🔲 Idle 📕 Poor 🛑 Ok 📳 Ideal 📕 Over		find_hotspots.exe! <u>initialize 2D buffer</u> – fi…
82	void initialize_2D_buffer (unsigned int mem_array [], unsigned int *fill_value	24		find_hotspots.exe! <u>render_one_pixel</u> +0xdf ····
83	4			find_hotspots.exe! <u>draw_trace</u> +0xc9 - find
84	<pre>// First (slower) method of filling array</pre>			find_hotspots.exe! <u>thread_trace</u> +0x/a = fi***
85	// Array is NOT filled in consecutive memory address order			find_hotspots.exe! <u>trace_shm</u> +Uxb3 - trac····
86	/**********************************/			find_hotspots.exe! <u>trace_region</u> +0x88 - tr····
87	<pre>for (int i = 0; i < mem_array_i_max; i++)</pre>	0.5%		find_notspots.exe! <u>renderscene</u> +0x49 - r···
88	{			find_notspots.exe! <u>rt_renderscene</u> +0x19
89	<pre>// Try to defeat hardware prefetching by varying the stride</pre>			find hotspots exeltbread uideo+0x8 = winter
90	<pre>int j(0), iteration_count(0);</pre>	0.1%		KERNEL 32 DI L'Base Thread Init Thunk+0x····
91	do {			ntdll dll/func@0x4b2e05da+0x2e = [unkno···
92	<pre>mem_array [j*mem_array_i_max+i] = *fill_value + 2;</pre>	22.9%		ntdll.dll <u>func@0x4b2e05b9</u> +0x1a - [unkno…
93	<pre>// Code to give the array accesses a non-uniform stride to defeat</pre>	E	=	
94	if ((iteration_count % 3) == 0) j=j+3;	23.1%		
95	<pre>else j=iteration_count;</pre>	0.7%		
96	iteration count++;			
97	Initialize 2D buffer 関数はプログラムの	8.5%	_	ľ
99 10	実行時間で合計 45% 余りを占める			



Threading 解析

各スレッドの動作状況を中心に解析 特に OpenMP*の利用について 有益な情報を表示する



シリアル/OpenMP* 領域での実行時間 (Elapsed Time)の割合と、OpenMP* 領域で 効率化できる割合 (Potential Gain)



🕑 Open MP Region CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously in an OpenMP region. Spin and Overhead time adds to the Idle CPU usage value. OpenMP regions in the drop-down list are sorted by Potential Gain (Elapsed Time) so it is recommended to start exploration from the top.





OpenMP* アプリケーションの パフォーマンス向上のヒント

OpenMP* 並列領域内の潜在的なパフォーマンス向上の可能性を評価します

 ロード・インバランス、スレッド生成、スケジューリング、ロックによるオーバー ヘッド

Grouping を [OpenMP* Region > Function > Call Stack] などに設定

Grouping: OpenMP Region / Function / Call Stack										
OpenMP Region / Function / Call Stack		Ope	enMP Potentia	al Gain 🔻		**		Number of OpenMP threads		
		Lock Contention	Creation	Scheduling	Reduction	Atomics	Elapsed Time		Instance Count	Idle
▶ primes\$omp\$parallel:64@/home/xlsoftkk/workspaces/takeda/ips/omp/prime.cpp:62:62	8.914s	0s	0s	0s	0s		18.873s	64		637.1
matmul\$omp\$parallel:64@/home/xlsoftkk/workspaces/takeda/ips/omp/matmul.cpp:25:34	0.026s	0s	0s	0s	0s	0s	0.055s	64	1	0.46
[Serial - outside parallel regions]							2.451s			1.44
matmul\$omp\$parallel:64@/home/xlsoftkk/workspaces/takeda/ips/omp/matmul.cpp:33:39							0.368s	64	1	22.4



マイクロアーキテクチャーの最適化

- Microarchitecture Exploration
- memory-access
- HPC Performance Characterization



これらの解析タイプは、CPU、メモリーの利用状況に深く 結びついた解析結果を表示します



プロセッサー上で実行されるプログラム

- プロセッサーは命令を実行する 主に実行される命令
- □ □ード/ストア
 - データをメモリーからレジスターへ読み込み
 データをレジスターからメモリーへ書き込み
- □ レジスターを使用して演算
 - ・ 四則演算、ビット操作、比較
- □ ジャンプ
 - ・ 次に読み込むプログラムの位置を変更
 - ・ 関数呼び出し/復帰や条件分岐を実現





コア内部の実行の流れ

コア内部で実行される命令は、複数の 独立した工程 (ステージ) を通して処理します

- フェッチ (Instruction Fetch)
 - メモリーから命令を取り込む
- デコード (Instruction Decode)
 - 取り込まれた命令を実行ユニットが読解可能な マイクロオペレーション (uOP) に変換する
- 実行 (Instruction Execution)
 - デコードされたマイクロオペレーションを実行する

リタイア (Instruction Retirement)

• 実行結果をメモリーにアップデートして命令の実行を完了する



CPU サイクルは止まらないが 命令の実行は止まる

パイプラインのスムーズな処理の流れを止める現象





パイプラインの最適化

無駄になったサイクルは実行が最適化されていない

・ 無駄になったサイクルを発見することが重要

命令あたりのサイクル数 (CPI) を確認して、 ストールしている個所に目安をつける

✓ CPI 値が高くなる主な要因

- ・実行コストの大きな命令
 - ・ 例:浮動小数点数の除算
- キャッシュミスを起こしたメモリーアクセスが多い
- ・頻繁な分岐命令の実行
 - ・ 分岐命令に 売く命令は、分岐方向が定まるまで未確定

メモリー階層

✓ キャッシュメモリーの存在

メモリーアクセスの待ちを低減するため の高速メモリー 一度アクセスしたデータを保持する 同じデータが再利用されるときに有効

主要なインテル[®] プロセッサー は L3 (LLC)、 L2、 L1 レベルのキャッシュ領域 を持っている

L2、L1 はコアごとに持ち、L3 はコアサ 有の領域として扱う 主要なインテル® プロセッサーのメモリー構造

コアヘデータ転送に要する時間と容量

コアの実行効率やメモリーアクセスの問題を調査する

Microarchitecture Exploration 解析

Microarchitecture Exploration	Microarchitecture Exploration	• (?)		
		Occurrent Districtions		
Analysis Configuration Collection Log	g Summary Bottom-up Event	Count Platform	ハイフラインの実行状が	んを祝見化
Selapsed Time ^[®] : 177.	312s 🖆			
Clockticks: 1	58,306,400,000		Issue: A significant	
Instructions Retired: 1	04,288,400,000		portion of Pipeline Slots is	
CPI Rate [®] :	1.518 💌	30.80% - Front-End	remaining empty due to	
MUX Reliability ^③ :	0.979	Bound	issues in the Front-End.	
	38.1% 🖻 of Pipeline Slots			
③ General Retirement [®] :	24.5% of Pipeline Slots		The metric value is high.	
Microcode Sequencer [®] :	13.6% 🕅 of Pipeline Slots	18.11% - Memory Bound	This can indicate that the	
	30.8% 🕅 of Pipeline Slots			
Front-End Latency ⁽⁹⁾ :	22.9% 🕅 of Pipeline Slots			
Front-End Bandwidth [®] :	7.9% of Pipeline Slots	38.06% - Petiring		
⊘ Bad Speculation ⁽²⁾ :	5.2% 🎙 of Pipeline Slots	36.00 % - Keuning		
Branch Mispredict [®] :	3.6% of Pipeline Slots			
Machine Clears ⁽²⁾ :	1.6% of Pipeline Slots			
Sack-End Bound [™] :	26.0% 🕅 of Pipeline Slots		/	
Memory Bound ⁽²⁾ :	18.1% 🕅 of Pipeline Slots		r	
S Core Bound ⁽²⁾ :	7.8% of Pipeline Slots		uPipe	
Total Thread Count:	3	This diagram represents ineffic	ciencies in CPU usage. Treat it as a pipe with an output flow	
Paused Time ⁽²⁾ :	Os	equal to the "pipe efficiency	/" ratio: (Actual Instructions Retired)/(Maximum Possible	
		Instruction Retired). If there a	are pipeline stalls decreasing the pipe efficiency, the pipe	
			shape gets more narrow.	

パイプライン内部の実行を視覚化 実行効率が^亜い状態では、カテゴリされている要因に 制限されてパイプは狭くなる

リタイア

できるだけ多くのスロットがこのカテゴリーになるようにする (ただし、このカテゴリーであってもさらに最適化できることがある)

不正なスペキュレーション

分岐予測ミスによるキャンセルなどにより、命令がリタイアせずにバックエンド から削除 (キャンセル) された場合

バックエンド依存

バックエンドにデータや長い実行を待機中の命令が含まれているため、フロントエンドが命令を供給しても、バックエンドがそれを受け取ることができない場合

フロントエンド依存

バックエンドは μop を受け取る準備ができているが、コードのフェッチや命令のデ コードの遅延によりフロントエンドが μop を供給できない場合

HPC アプリケーションに有益な情報を提供

HPC Performance Characterization 解析

HPC アプリケーションの最適化に重要な情報を提示します・ CPU 利用状況、メモリー帯域、FP ユニットの活用...

✓ FPU 負荷の比率 (%) (FPU に完全に負荷がかかっている場合は 100%、しきい値は 50%)
 ✓ FPU 使用率による上位 5 つのループ/関数

スカラーとパックドに GFLOPs を細分化 (ベクトル化されているかどうか)

ベクトル化の最適化にはインテル[®] Advisor

スレッドのプロトタイプ生成およびベクトル化の最適化を支援するツール

ベクトル化アドバイザーによる情報収集が効果的

ループがベクトル化 されているか? ロペクトル化されたループ ペクトル化されなかったループ 次に			フォーマンスを げているものは? こすべきことは?		ループに 費やされた 時間		最新の命令セットを 利用しているか?								
							ベクトル化を 妨げているものは?		ベクトル化の効率			ж К			
Ē 9	🗉 Sum 🛯 ary 🛯 Survey & Roofline 📲 Refir				neme	nt Report									
찡	+	- Function Call Sites and Loops			0005		@ Performance	Self Time - Total	Total Time	Type	Why No	ector	rized op	S	
- PFL		Tunction can sites and coops			Issues	Sen fille +	Total fille	туре	Vectorization?	Vect	Efficien	Gain E			
N.	= 0	[loo	o in main at r	oofline.cp	op:247]		2 Ineffective pe	7.594s 🗖	7.594s	Vectorized (Bod		AVX2	<mark>3</mark> 1%	1.22x	
	(۵ <mark>ا] گ</mark> 🛙	oop in main at	t roofline.c	cpp:247]		I Possible ineffici	7.516s 🗖	7.516s 🛙	Vectorized (Body)		AVX2			
		۵ <mark>ا گ</mark> 🗉	oop in main at	t roofline.c	cpp:247]			0.078s1	0.078s1	Remainder					
	+ ୯	[loop]	o in main at ro	ofline.cpp	p:260]		I Ineffective peel	3.016s	3.016s I	Vectorized (Body		AVX2	99%	3.98x	
	+ ୯	[loop]	o in main at ro	ofline.cpp	o:273]		I Ineffective peel	2.484s 🛙	2.484s	Vectorized (Body		AVX2	99%	3.98x	
	۵ 🕐	[loop	o in main at ro	ofline.cpp	:256]			0.016s1	3.031s1	Scalar	inner loop				•
	•				•									•	

インテル[®] VTune Profiler を 活用する

例えば、CPU 利用率にバラつきがある

全ての CPU コアが利用されているが、各コアの使用率にバラつきがある

スレッド毎に異なる処理を実行していること もあるので、意図した動作なのか見る

全スレッドを使用した計算主体の 処理である場合、大きく最適化できる可能性

最適化の余地を調査する

Hotspots 解析からホットスポットを検索する

• ユーザーコードにフィルタリング

シリアル処理の場合、マルチスレッド化による複数コアの利用が可能か検討する

マルチスレッド化されている場合は、Timeline の内容に注目

• CPU 利用率が少ない場合、効率よく複数のスレッドによる処理が行われていないことを示しています

Threading 解析から制限する要因を確認します

• 複数のスレッドを扱う処理に注目

Hotspots 解析の実行

(OpenMP で並列化されている)

次のステップ→ Bottom-up を確認

タイムラインからスレッドの利用効率が 低い処理を特定

	Grouping: Function / Call	l Stack						
			CPU Time 🔻					Viewing < 1
	Function / Call Stack		Effective Time by U Idle B Poor Ok	Spin Time Overhead Time		Mo	dule 97. matmul-prin	
	FindPrimes\$omp\$1	8	4.671s		0s	0s	matmul-pri	ne-pi.exe VCOMP140
	NtYieldExecution		0s	-	0.528s	0s	ntdll.dll	VCOMP140
indPrimes	問数の実行	二問題	iがありそう		0.078s	Os	ntdll.dll	VCOMP140
indi innes					0.031s	Os	KERNELB	ASE.dll
	GetTickCount		Os		0.016s	Os	KERNEL32	2.DLL
		対象	タイムライ 肉区間で実	ンでフィル 行された	ノタリン? 処理が上	グすると、 _に表示さ	れる	
	ρ: +	対象	タイムライ えの区間で実	ンでフィル 行されたり 10s	レタリン: 処理が上	グすると、 に表示さ	れる 25s	30s
	ৃ : + ৮ ৮ [unc@0x1800080eb (対象 TID: 6 ¹⁵²⁾	<mark>タイムライ</mark> の区間で実 ^{03 55}	ンでフィル 行された ^{10s}	レタリン? 処理が」 ^{15s}	グすると、 こに表示さ	れる 25s	30s
	D: + func@0x1800080eb (func@0x1800080eb (対象 TID: 6 52) TID: 1 2264)	<mark>タイムライ</mark> の区間で実	ンでフィル 行されたタ 10s	レタリン 処理が上 15s	グすると、 こに表示さ 205	れる 25s	30s
	Ø: + func@0x1800080eb (func@0x1800080eb (func@0x1800080eb (大方 (TID: 6 952) (TID: 1 2264) (TID: 1 260)	タイムライ の区間で実 0s 5s	ンでフィル 行された ¹⁰⁵	<mark>ノタリン</mark> ク 処理が上	ブすると、 こに表示さ	<mark>れる</mark> 25s	30s
	Ø: ↓ geget func@0x1800080eb (func@0x1800080eb (func@0x1800080eb (func@0x1800080eb (func@0x1800080eb (大寸多 TID: 6 52) TID: 1 2264) TID: 1 260) TID: 7 372)	タイムライ の区間で実 0s 5s	ンでフィル 行されたタ 10s	レタリン 処理が上 15s	^{びすると、} に表示さ	れる 25s	30s
	D: + func@0x1800080eb (func@0x1800080eb (func@0x1800080eb (func@0x1800080eb (func@0x1800080eb (func@0x1800080eb (大寸多 (TID: 6 52) (TID: 1 2264) (TID: 7 372) (TID: 1 712)	タイムライ の区間で実 □s 5s	ンでフィル 行された ¹⁰⁵	レタリン 処理が上 15s	びすると、 こに表示さ 20₅	れる 25s	30s
	Ø: + func@0x1800080eb (大寸 身 TID: 6 52) TID: 1 264) TID: 7 72) TID: 7 72) TID: 1 712) TID: 1 712)	タイムライ ② 区間で実 ③ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □	ンでフィル 行された ¹⁰⁵	レタリン 処理が」	グすると、 に表示さ	れる 25s	30s
	𝔅 : ➡ gevent func@0x1800080eb (func@0x1800080eb (func@0x1800080eb (大寸 倉 TID: 6 52) TID: 1 2264) TID: 1 2264) TID: 1 260) TID: 7 72) TID: 1 712) TID: 1 52) TID: 4 44)	タイムライ の区間で実 0s 5s 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	ンでフィル 行された 10s	<mark>ノタリン</mark> 処理が上	ブすると、 こに表示さ 20s	れる 25s	30s

次のステップ→ 「hreading 解析を実行

Threading 解析から FindPrime に注目

コードの確認

並列領域内のループ処理で、仕事量のバランスが華い状態

🛦	Source	🖕 CPU Time: Total	Effective Time by Utilization 🔊
17	end = Max_Numbers;		
18			
19	<pre>#pragma omp parallel for private(factor) reduction(+:PrimeCount)</pre>	2.0%	Os
20	for (number = start; number < end; number += stride)		
21	{		
22	factor = 3;		
23	while ((number % factor) != 0)		
24	factor += 2;	90.1%	144.481s
25	if (factor == number)		
26	{		
27	PrimeCount++;		
28	}		
29	}		
30	return 0;		
31	}		

ロードバランス (処理の均一性)の維持

各スレッドの処理量を均一化する OpenMP* では右図のような 状況が発生しやすい

#pragma omp parallel for
for (int i=0; i<10000; i++){
 for(int j=0; j<max; j++){
 if(...) break;
 }</pre>

#pragma omp parallel for schedule(static, 10)

schedule 句: ループの分割の 大きさと割り当て方の方針を指示する

考えられる異なるケース

仕事量の差の他にも、同期処理によって待機が発生 していることも

最適化の余地を調査する – もうー度掲載

Hotspots 解析からホットスポットを検索する

ユーザーコードにフィルタリング

シリアル処理の場合、マルチスレッド化による複数コアの利用が可能か検討する

マルチスレッド化されている場合は、Timelineの内容に注目

• CPU 利用率が少ない場合、効率よく複数のスレッドによる処理が行われていないことを示しています

Threading 解析から制限する要因を確認します

• 複数のスレッドを扱う処理に注目

XIISOFT

Threading 解析を実行

次のステップ→ Bottom-up を確認

ロック・オブジェクトに注目

クリティカル処理は draw_task::operator() 関数からコールされている

ソースコードを確認

Syne Object / Function / Can Otack	📕 Idle 📕 Poor 📒 Ok
Critical Section 0x6e8d14dd	49.069s
draw_task::operator()	42.951s
[TBB parallel_for on class draw_task]	6.117s 📒
Auto Reset Event 0xf5b5da75	13.772s

draw_task::operator() 関数をダブルクリック

クリティカル・セクション内部の処理に注目

- 本当に必要なクリティカル処理か
- atomic 処理に置換できないか
- 並列領域外に出せないか

<pre>void operator () (const tbb::blocked_range <int> &r)</int></pre>	
unsigned int serial = 1;	
pthread mutex lock() によるクリティ	[・] カル・セクションに
	が確認できます
多くの CPU 时间を 府員していること	が推認できみり
<pre>for (int y=r.begin(); y!=r.end(); ++y) {</pre>	
drawing_area drawing(startx, totaly-y, stopx-	-
// Acquire mutey to protect pixel calculation	r
<pre>pthread_mutex_lock (&rgb_mutex);</pre>	42.951s
<pre>for (int x = startx; x < stopx; x++) {</pre>	
<pre>color_t c = render_one_pixel (x, y, local</pre>	1
drawing.put_pixel(c);	
}	
<pre>// Release the mutex after pixel calculation</pre>	
<pre>pthread_mutex_unlock (&rgb_mutex);</pre>	
if(!video->next_frame()) tbb::task::self().ca	
}	
}	

排他制御による遅延

排他制御を最小限にする

スレッド番号

並列数が多いほど影響も大きい 同期処理自体の大きさ、実行回数と頻度に注意

CPU 使用率の高いプログラム

実行に時間は掛かっているが、CPUの各コアの使用率は高い 一見プログラムは十分にコアを活用しているように思われるが・・・・

最適化の余地を検証する

CPU コアの利用は正常にアプリケーションの計算/処理に充てられているのか?

▶ 余分な処理に費やされていないか確認する

スレッドの待機によるものではない場合、Threading 解析では、特定が 難しいことも

Hotspots 解析を実行して、パイプライン中の実行に 問題が潜んでいないか確認

パイプラインの実行効率が低い場合は、Microarchitecture Explorationを実行して、どの段階で実行が制限されているか確認

Hotspots 解析の実行

	=		
Hotspots Hotspots by CPU Utilization 👻 🕐	INTEL VTUNE AMPLIFIER 2019		
Analysis Configuration Collection Log Summary Bottom-up Caller/Callee Top-down Tree Platform			
 Elapsed Time[®]: 49.434s CPU Time[®]: 350.273s Instructions Retired: 295,784,539,322 Microarchitecture Usage[®]: 11.1% k of Pipeline Slots 	Hotspots Insights If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee view to track critical paths for these hotspots.		
CPI Rate [®] : 4.379 N Wait Rate [®] : 5.369 低い Microarchitecture Usage に注目 12 Paused Time [®] : 0.122s	Explore Additional Insights Microarchitecture Usage ⊙ : 11.1% ► Use Microarchitecture Exploration to explore how efficiently your application runs on the used hardware.		

✓ Top Hotspots

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

Function	Module	CPU Time [®]
multiply1	matrix.exe	349.775s
func@0x1401b9c20	ntoskrnl.exe	0.109s
init_arr	matrix.exe	0.054s
func@0x1c0006610	NETIO.SYS	0.041s
malloc	ucrtbased.dll	0.038s
[Others]		0.256s

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

Microarchitecture Exploration 解析の 実行が推奨されている

次のステップ→ Bottom-up から関数レベルの情報を確認

Microarchitecture Usage 項目を確認

B 좌 ▶ 호 O ▷ ♡	Welcome 🗙 r000hs	×			=	
Motspots Hotspots	by CPU Utilization 🝷 🔇	D	mul	tiply	1 関数の Microarchitecture	
Analysis Configuration Co	llection Log Summary	Bottom-up Caller/Callee T		upty	T EXECUTE CITE CITE CITE	
Grouping: Function / Call Stac	ck			Us	age が低くなっている	
	>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>					
Function / Call Stack	CPU Time 🔻	Instructions Retired Microarch	nitecture Usage	Wait Rate	99.7% (348.758s of 349.775s)	
multiply1	349.775s	295,204,614,781	11.1%	1.099	matrix.exelmultiply1 - multiply.c	
Tunc@0x1401b3c20	0.1055	121,244,740	3.078	0.000	matrix.exel nread-unction+Ux1/1 - thrmodel.c.57	
▶ init_arr	0.054s	174,226,096	28.3%	0.029	Netless2. United to the second s	
▶ func@0x1c0006610	0.041s	0	0.0%	0.000		
▶ malloc	0.038s	45,238,321	21.5%	0.279	1	
▶ func@0x1401093d4	0.027s	39,305,320	10.7%	0.000		
▶ func@0x1401b4d90	0.017s	0	20.3%	0.000		
▶ func@0x140049fb0	0.016s	38,127,672	22.0%	0.000		
▶ free	0.016s	0	16.0%	0.000		
KeAcquireInStackQueuedS	0.016s	0	0.0%	0.000		
▶ func@0x1c0007810	0.016s	0	21.8%	0.000		
▶ func@0x1c0034a1b	0.016s	0	0.0%	0.000		
KeSetEvent	0.015s	0	0.0%	0.000		
Kel owerlral	0.015c	20 18/ 522	0.0%	0 000		
Ø: + − ⊭	jrr 0s 5s	10s 15s 20s	25s 30	s 35s	40s 45s 🕜 Thread 🔻	
ਲ Thread (TID: 10496)	i Mirindi i					
Thread (TID: 7204)					Context Switches	
Thread (TID: 4100)	and the second second	print a state of the			Synchronization	
Thread (TID: 7876)		A AND A A			CPU Time	
Thread (TID: 2744)					Spin and Overhead	J
Thread (TID: 8632)				1.	次のステップ→	
Thread (TID: 1936)		A CONTRACTOR OF A CONTRACT				
Thread (TID: 9316) CPU Time					Microarchitecture Exploratio 解析の実行	n
FILTER 100.0%	Any Process 🔻	Any Thread Any Modu	Ile 🔻 Any Utili	zatic 🔻 🛛 U	Jser functions + 1 ・ Functions only 月午初の天1」	

Microarchitecture Exploration 解析の実行

Seffective Physical Core Utilization[™]: 97.4% (3.896 out of 4)

Effective Logical Core Utilization 2: 97.2% (7.775 out of 8)

⊘ Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spir

次のステップ→ LLC Miss が発生している関数を確認

LLC のミスが多発している処理を特定

~~								
37	c[i][j] = c[i][j] + a[i][k] * b[k][j];							
38	}							
39		スルー	- プ加理					
40	Induipty) 與 数 C 关 表 C 1 C C ·	212	ノル注					
41	一方式の							
42								
43	void multiply1 (int msize, i							
4.4								
45	int i,j,k;							
46								
47	// Naive implementation							
48	<pre>for(i=tidx; i<msize; i="i+numt)" pre="" {<=""></msize;></pre>							
49	<pre>for(j=0; j<msize; j++)="" pre="" {<=""></msize;></pre>	0.0%						
50	for(k=0; k <msize; (<="" k++)="" td=""><td>0.0%</td><td>0.5%</td><td>0.1%</td></msize;>	0.0%	0.5%	0.1%				
51	c[i][j] = c[i][j] + a[i][k] + b[k][j];	0.1%	79.1%	4.8%				
52	}	0.0%	0.7%	0.0%				
53	}							
54	}							
55	}							
56	void multiply2(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM], TYPE c[][NUM],							

対策:メモリーへのアクセス頻度を落とす ≒ キャッシュのヒット率を上げる

- ここでは配列のアクセス順序の変更が効果的 → ループ処理の順序を入れ替え 他にも…
- **ループのアンロールを適用** ← コンパイラーオプションで調整
- ・ フォルスシェアリングが発生しているか ← 解析結果の False Sharing 項目を確認
- ・ NUMA 構成の場合、アフィニティを決定してみる ← Memory Access 解析も実行

まとめ

- ✓ 最適化にはホットスポットの特定と、ボトルネックの有無を調査する ことが重要です。
- ✓ インテル[®] VTune Profiler はさまざまなシナリオにおける、 最適化作業で使用いただけます。
- ✓ 予め用意された解析タイプを使用すると、最適化を進めるために有益な情報を簡単に取得することができます。

インテル[®] VTune[™] Profiler 30日間お試し版 <u>https://www.xlsoft.com/jp/products/download/intelj.html</u>

インテル[®] VTune[™] Profiler 日本語化

iSUS にて日本語化パッケージが公開されています!

https://www.isus.jp/products/vtune/vtune_jp/

🍻 解析の設定	INTEL VTUNE AMPLIFIER 201	9
■ ローカルホスト …	えるように	
● ● アプリケーションを起動 ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ●	 アブリケーションのパフォーマンスに影響する CPU マイクロアーキテクチャーのボトル ネックを解析します。この解析タイプは、ハードウェア・イベントベース・サンプリン グ応集を使用します。 副連問 (CPU サンプリング開稿(ミリ秒) ② 投機の問題^①: ○ バックエンド依存^①: 	0.1% パイプライ 85.1% ▶ パイプライ
解析ターゲットの指定と設定: 実行するアプリケーションまたはスクリプト アプリケーション: C:\Users\xlsoftkK\Desktop\matrix\vc15\x64\Debug\matrix.exe	上位レベルのメトリックの粒度を マ フロントエンド依存 マ 投機の問題 ロ バレレ 佐存 □ 1 低存 □ 2 低存 □ 2 し 1 低存 □ 2 低存 □ 2 し 1 低存 □ 2 低存 □ 2 □ 2 トエンド依存	80.1% ▶ パイプライ ロックテ・ バレキャッシュ) は、メインメモリー (DRAM) 階層 、最後の、そして最も長いレイテンシーのレベル
 アプリケーションの引数: アプリケーション・ディレクトリーを作業ディレクトリーとして使用 作業ディレクトリー: C:\Users\xlsoftkk\Desktop\matrix\vc15\x64\Debug ご 高度 ▶ 	 ◇ メモリー報存 ◇ コア依存 ◇ リタイア ◇ メモリー帯域幅を解析 ◇ Evaluate max DRAM band wgモード ◇ メモ ◇ メモ ◇ レロー報道 ◇ マーボー ◇ マーボー ◇ レローボー ◇ レ	のメモリー要求ミスは、レイテンシーの長いロー モート DRAM によって処理されます。LLC ミス 、すべてのサイクルに対する LLC ミスが未処理の 率です。 ンプ ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・
100× .	詳細	

インテル[®] VTune[™] Profiler オンラインヘルプ (iSUS 翻訳版) <u>https://www.isus.jp/products/VTune[™]/help-top/</u> Intel 社 Web 版 (英語) <u>https://software.intel.com/en-us/vtune-Profiler-help</u>

インテル[®] VTune[™] Profiler チュートリアル (英語) https://software.intel.com/en-us/articles/intel-vtune-Profiler-tutorials

プロセッサー世代別最適化ガイド(英語)

https://software.intel.com/en-us/articles/processor-specific-performance-analysispapers

インテル[®] VTune[™] Profiler パフォーマンス解析クックブック

<u>https://jp.xlsoft.com/documents/intel/vtune/2019/Intel_VTune_Profiler_Performan</u> <u>ce_Analysis_Cookbook.pdf</u> Intel 社 Web 版 (英語)

https://software.intel.com/en-us/vtune-Profiler-cookbook

インテル[®] VTune[™] Profiler XE の General Exploration (一般解析) がどのように動作する かを理解する - iSUS

https://www.isus.jp/products/vtune/understanding-how-general-explorationworks-in-vtune/

補足情報

サンプリング・ドライバーの動作確認

Windows* ターゲット

• amplxe-sepreg.exe -s の実行

C:¥Program Files (x86)¥IntelSWTools¥VTune Amplifier 2018¥bin64>amplxe-sepreg.exe -s Checking sepdrv4_1 driver path...OK Checking sepdrv4_1 service... Driver status: the sepdrv4_1 service is running Checking sepdal driver path...OK Checking sepdal service... Driver status: the sepdal service is running Checking socperf2_0 driver path...OK Checking socperf2_0 service... Driver status: the socperf2_0 service is running

サンプリング・ドライバーがロードされているか確認

配置先:C:¥Program Files (x86)¥IntelSWTools¥VTune Profiler¥bin32

サンプリング・ドライバーの動作確認

./insmod-sep -q の実行

配置先:/opt/intel/VTune_Profiler/sepdk/src

[xlsoftkk@knl src]\$./insmod-sep -q
pax driver is loaded and owned by group "vtune" with file permissions "666".
socperf2_0 driver is loaded and owned by group "vtune" with file permissions "666".
sep4_1 driver is loaded and owned by group "vtune" with file permissions "666".
vtsspp driver is loaded and owned by group "vtune" with file permissions "666".

サンプリング・ドライバーがロードされているか確認

バージョン 2018 Update1 以降では ./amplxe-self-checker.sh を実行

 適切にドライバーがインストールされ、システムがパフォーマンス・データを 収集できるか検証するセルフチェック・スクリプト

配置先: /opt/intel/VTune_Profiler/bin64

解析を始める前に

アプリケーションの準備

Visual C++*、GNU* gcc、およびその他の C++ コンパイラー、および Fortran コンパイラーで生成されたバイナリーを解析することができます

インテル[®] コンパイラー以外でもOK

ソースコードと関連付けて解析を行うためシンボル情報を生成します

- ・ デバッグ情報の付加
 - Windows*: /Zi
 - Linux*: -g
 - デフォルトの最適化 (/O2 および -O2) が無効になるため、明示的に指定し直す

デバッグ情報がないとループや関数のソースとの関連付けができません

インテル[®] コンパイラー利用時に 設定したいオプション

追加のオプションを指定することで、解析結果により多くの関数、ループ、 ソース情報を追加できます

関数のインライン展開用デバッグ情報の追加 (icc/ifort)

• Windows*: /debug:inline-debug-info , Linux*: -debug inline-debug-info

OpenMP* 利用時のソースコード情報 (icc/ifort)

• Windows*: /Qparallel-source-info=2 , Linux*: -parallel-source-info=2

インテル[®] スレッディング・ビルディング・ブロック (インテル[®] TBB) を使用している場合 Windows*: /DTBB_USE_THREADING_TOOLS Linux*: -DTBB_USE_THREADING_TOOLS

データ収集の対象となる範囲を制御する

API によりソースコードレベルの解析を制御できます

➤Fortran、C/C++ コードが対象

主要な API 機能

- ・void __itt_pause (void): アプリケーションのデータ収集を一時停止します
- ・ void __itt_resume (void): データ収集を再開します
- void __itt_detach (void): データ収集をデタッチ (終了) します

```
使用ヘッダーとリンク・ライブラリー
Windows*:
<install_dir>¥include¥ittnotify.h
<install_dir>¥lib32¥libittnotify.libまたは <install_dir>¥lib64¥libittnotify.lib
Linux*:
<install_dir>/lib32/libittnotify.a または <install_dir>/lib64/libittnotify.a
```


MPI アプリケーションの解析

コマンドベースで解析を実行する

mpirun -n 16 -ppn 4 -l amplxe-cl -collect hotspots -trace-mpi -result-dir my_result -- my_app.out

- ・全ノードで解析を実行すると、各計算ノードごとの解析結果が表示されます
 - ・ 上記だと4つの解析結果を生成します
- -trace-mpi インテル[®] MPI ライブラリー以外が提供する MPI ランチャーから実行するときに指定

インテル[®] MPI ライブラリー (バージョン5.0.2以降) を使用している場合

mpirun -gtool "amplxe-cl -collect hotspots -result-dir my_result:7,5" my_app.out

-gtool オプションは MPI アプリケーションの解析に必要な設定を簡易化します

お問い合わせはこちらまで <u>https://www.xlsoft.com/jp/qa</u>

Intel、インテル、Intel ロゴは、アメリカ合衆国および /またはその他の国における Intel Corporation またはその子会社の商標です。 *その他の社名、製品名などは、一般に各社の商標または登録商標です。 インテル[®] ソフトウェア製品のパフォーマンス / 最適化に関する詳細は、<u>Optimization Notice (最適化に関する注意事項)</u>を参照してください。 © 2019 Intel Corporation. 無断での引用、転載を禁じます。

XLsoft のロゴ、XLsoft は XLsoft Corporation の商標です。Copyright © 2019 XLsoft Corporation.

