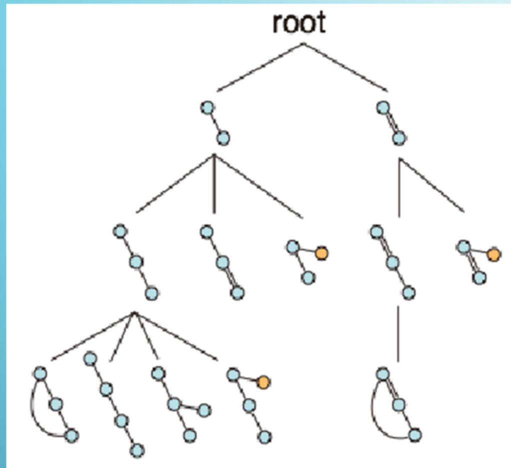
GPGPUを用いた頻出部分グラフマイニングの実装とプロファイリング

尾崎研究室　　土岐 達哉

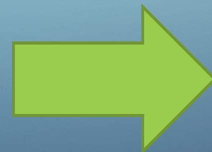
背景と目的



[AGM, *in PKDD '00*]
[FSG, *in IEEE Trans. on KDE*
[gSpan, *in ICDM '02*]
⋮

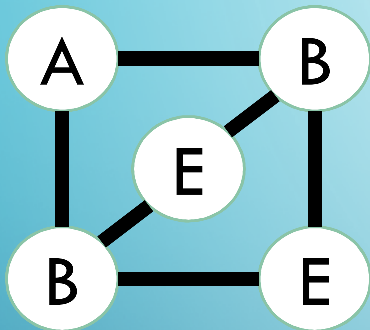
+

高速化

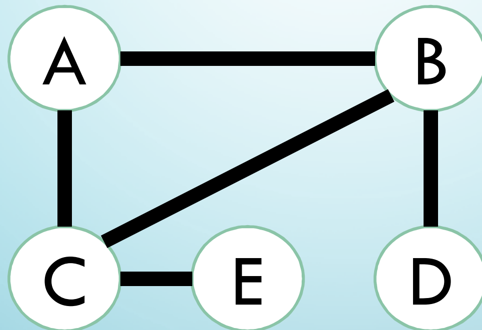


分析

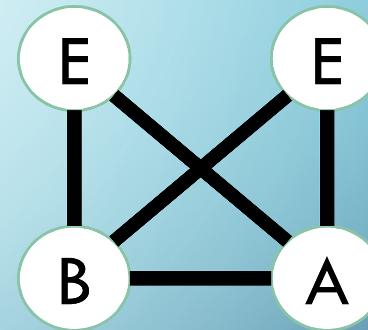
頻出部分グラフマイニング



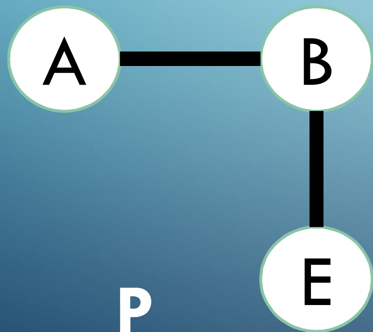
G1



G2



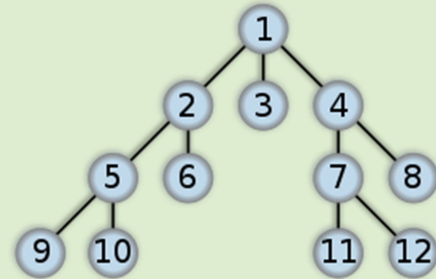
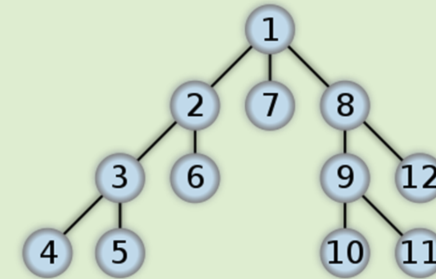
G3



P

最小支持度2の時，Pは頻出

→ 頻出するパターンを列挙

	幅優先探索	深さ優先探索
		
CPU	**AGM**, '00 **FSG**, '01 etc…	**gSpan**, '02 etc…
GPU	**Graph-Based Substructure Pattern Mining Using CUDA Dynamic Parallelism**, *in IDEAL '13* **Parallel Graph Mining with GPUs**, *in JMLR W&CP '14*	

今回は
このアルゴリズムを実装→

GPUとGPGPU

- GPU
  - PCで画像処理を担当する部品
  - どのPCにも入っている
- GPGPU
  - GPUの演算能力を画像処理以外に応用する技術

GPUの特徴 “コア数”

- CPUに比べ，コア数が非常に多い
  - CPUで一般的に高性能なものは16コア
  - GPUでは1664コア（GTX970の場合）
- 並列処理が得意

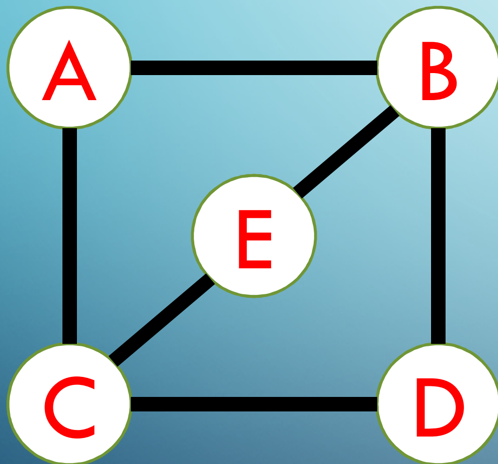
フリンの分類

GPU

データ	Single Instruction Multiple Data (SIMD)	Multiple Instruction Multiple Data (MIMD)
	Single Instruction Single Data (SISD)	Multiple Instruction Single Data (MISD)
	命令	

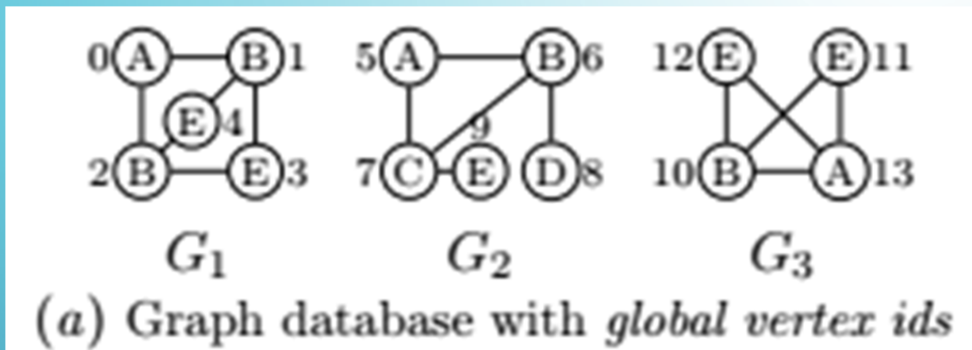
実装　“GPUを用いた並列化”

- 近傍を求める



- Parallel Primitives
  → 並列処理に適した演算
  → gSpanアルゴリズムから変換

実装 “データ構造”



(*a*) Graph database with *global vertex ids*

global id	0	1	2	3	4	5	6	7	8	9	10
N	1 2	0 3 4	0 3 4	1 2	1 2	6 7	5 7 8	5 6 9	6	7	11 12 13

global id	11	12	13
N(cont'd)	10 13	10 11 12	10 11 12

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
O	0	2	5	8	10	12	14	17	20	21	22	25	27	30	-1

(*b*) GPU representation (partial) of graph database

***Robert Kessl et al*, Parallel Graph Mining with GPUs**, *in JMLR W&CP '14*

- グラフを一次元の配列で表現
- GPU上では可変長の多次元配列が出来ない…

実装 “疑似コードの一例”

Algorithm 1 Graph Mining on GPUs (Database $\mathcal{D}$, Threshold $minsup$, Pattern P, Embeddings $\Sigma_{\mathcal{D}}(P)$)

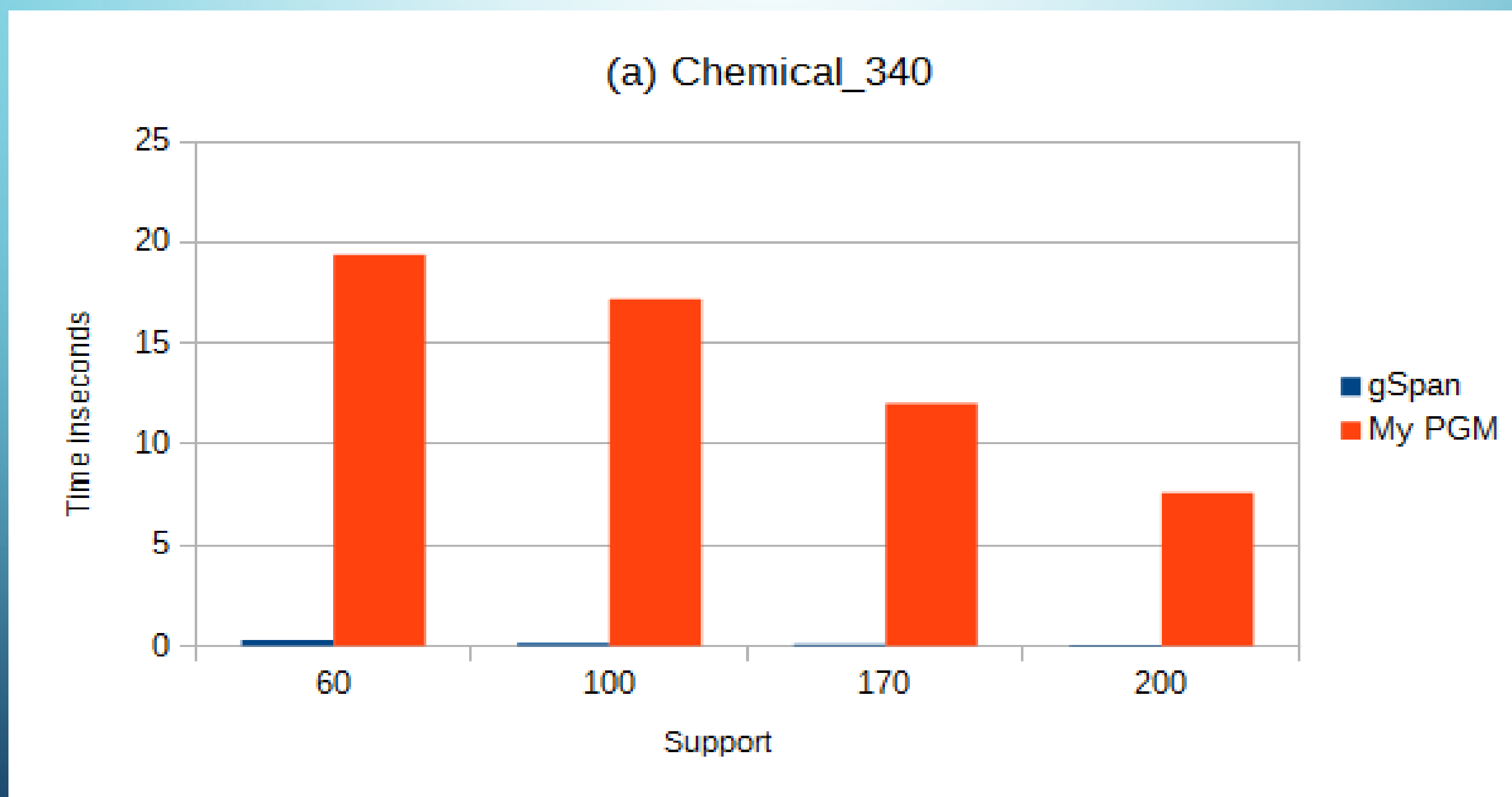
//Initial Call: $P = \emptyset$, $\Sigma_{\mathcal{D}}(P) = \emptyset$

1: (GPU step) Get all possible edge extensions $\mathcal{E}_{\mathcal{D}}(P)$ of P
2: (GPU step) Compute support for all extensions in $\mathcal{E}_{\mathcal{D}}(P)$ and remove infrequent extensions
3: **for** each $e = (v_i, v_j, l_i, l_{ij}, l_i) \in \mathcal{E}_{\mathcal{D}}(P)$ **do**
4: $P' \leftarrow P$ extended by e
5: **if** $P' = minDFS(P')$ **then** → 約100行
6: output P'
7: (GPU step) Create $\Sigma_{\mathcal{D}}(P')$
8: GRAPHMINING($\mathcal{D}$, $minsup$, P', $\Sigma_{\mathcal{D}}(P')$)
9: **end if**
10: **end for**

Robert Kessl et al, **Parallel Graph Mining with GPUs**, *in JMLR W&CP '14*

実験結果（gSpanとの比較）

世界的に有名な
高速アルゴリズム



実験結果

- CPU版に比べ，とても遅くなってしまった…
- 原因の特定が難しい
  - ライブラリを積極的に使ったことが原因？
  - 疑似コードが無い部分のコードが拙い？
- そうだ，プロファイリングしよう！

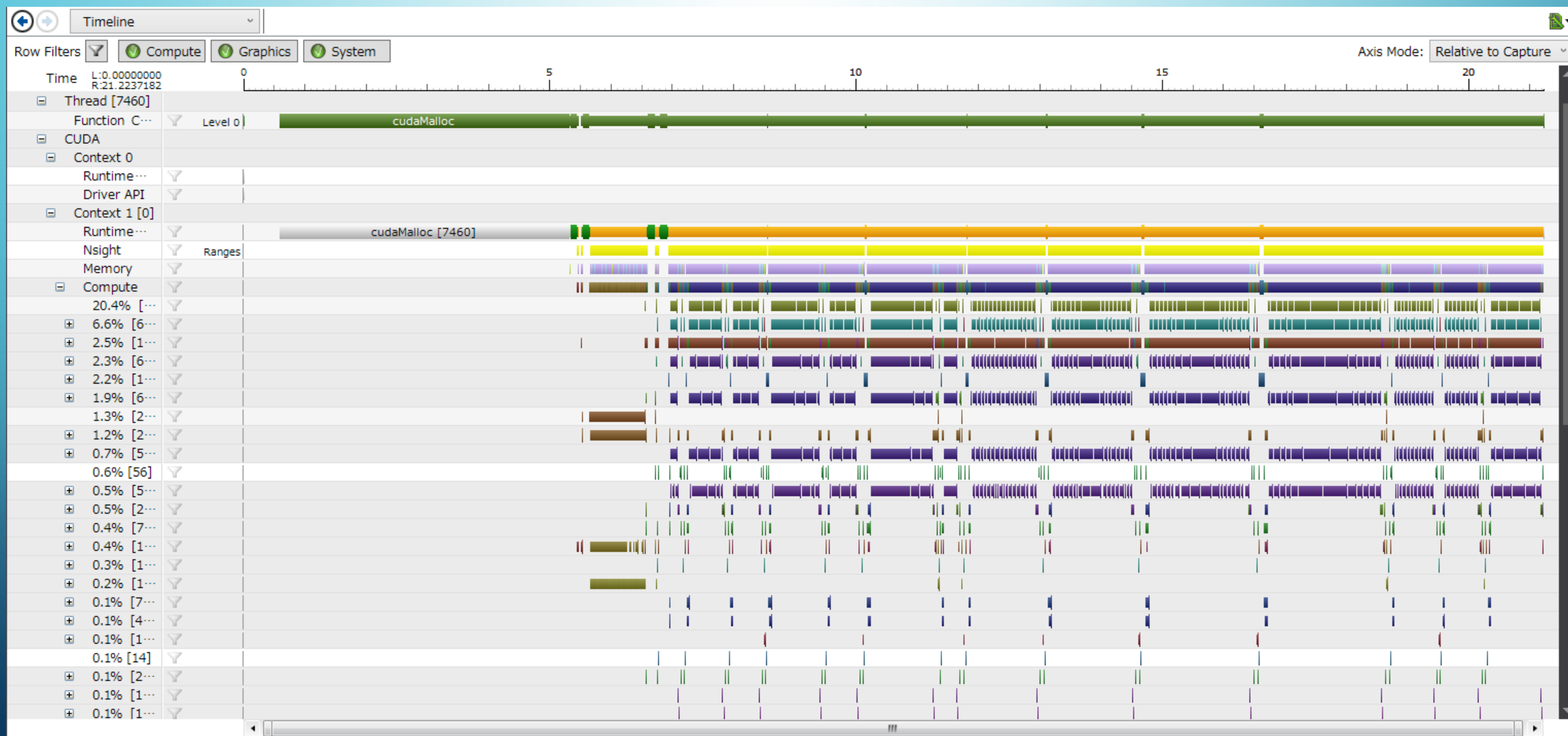
プロファイリング

- プログラムのプロファイリング
  ＝「アルゴリズムの各部分の消費時間を分析する」
- 時間のかかる悪い処理部分を特定する
  ≒犯罪者プロファイリング

Timeline分析

Nsight

（NVIDIA社が提供する公式プロファイラ）



時間のかかっている処理の特定

	Function Name	Grid Dimensions	Block Dimensions	Start Time (μs)	Duration (μs)
1	storeExt_kernel	{58, 1, 1}	{1024, 1, 1}	16,032,777.254	73,426.948
2	storeExt_kernel	{49, 1, 1}	{1024, 1, 1}	14,130,186.286	65,454.189
3	storeExt_kernel	{39, 1, 1}	{1024, 1, 1}	12,592,313.119	55,567.570
4	storeExt_kernel	{28, 1, 1}	{1024, 1, 1}	11,316,717.518	44,321.379
5	storeExt_kernel	{30, 1, 1}	{1024, 1, 1}	9,695,876.487	38,150.773
6	storeExt_kernel	{24, 1, 1}	{1024, 1, 1}	8,146,141.293	30,968.362
7	storeExt_kernel	{17, 1, 1}	{1024, 1, 1}	6,846,396.472	29,099.105
8	storeExt_kernel	{11, 1, 1}	{1024, 1, 1}	19,689,945.462	23,775.075
9	storeExt_kernel	{39, 1, 1}	{1024, 1, 1}	18,955,060.195	23,765.667
10	storeExt_kernel	{19, 1, 1}	{1024, 1, 1}	18,138,135.528	21,960.439
11	storeExt_kernel	{15, 1, 1}	{1024, 1, 1}	9,107,660.530	16,811.821
12	storeExt_kernel	{16, 1, 1}	{1024, 1, 1}	7,560,994.670	16,049.510
13	storeExt_kernel	{9, 1, 1}	{1024, 1, 1}	6,571,338.235	14,325.847
14		{10, 1, 1}	{768, 1, 1}	18,646,747.640	11,231.605
15		{10, 1, 1}	{768, 1, 1}	12,030,360.265	11,122.967
16		{10, 1, 1}	{768, 1, 1}	18,442,708.417	10,885.535
17		{10, 1, 1}	{768, 1, 1}	12,076,024.808	10,774.625
18		{10, 1, 1}	{768, 1, 1}	18,217,276.754	10,774.401
19		{10, 1, 1}	{768, 1, 1}	18,419,428.054	10,753.283
20		{10, 1, 1}	{768, 1, 1}	18,622,854.929	10,727.171

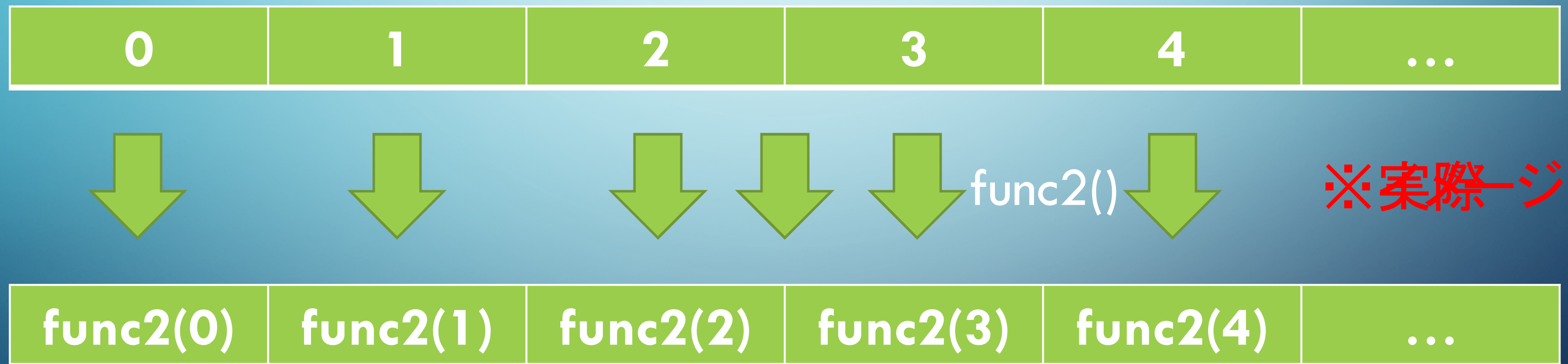
- 関数名の消失

(注)公式プロファイラと
公式ライブラリ

- ・storeExt_kernel()
  ・名無し関数()
  が遅い原因

原因の特定

- どちらにもtransform(array, func2())がある
- transform()もfunc2()も並列処理



並列処理のイメージ

transform			
0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

func2			
0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

func2			
0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

func2			
0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

・・・

考察と改善策

- 2段階&多数のスレッド起動がボトルネック
- func2()を並列でなく逐次処理にすれば改善？

改善策のイメージ

transform			
0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

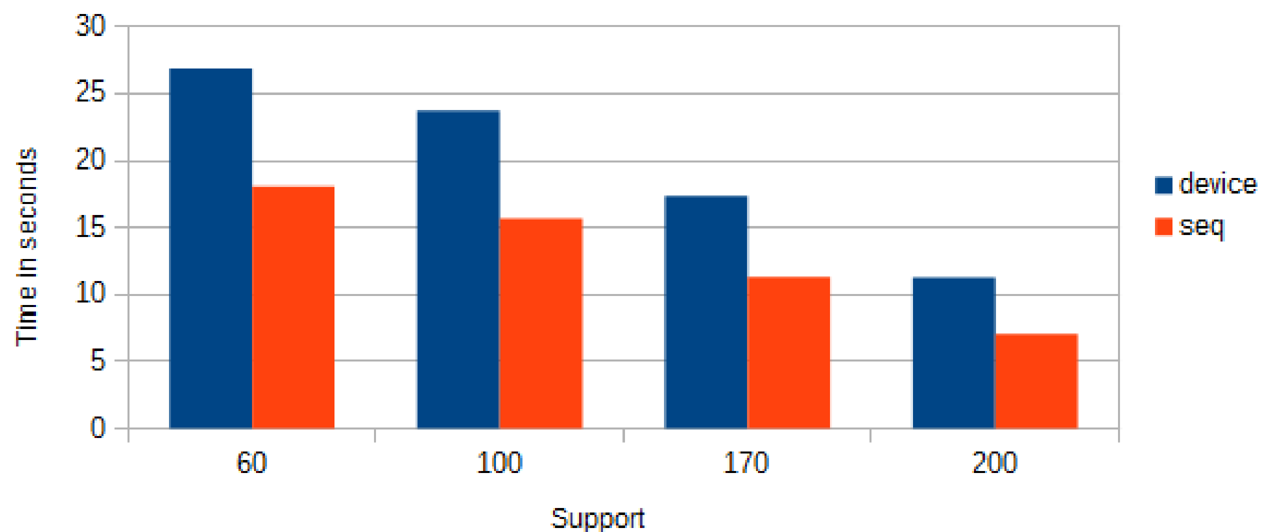
func2					
0	1	2	3	4	…

func2					
0	1	2	3	4	…

func2					
0	1	2	3	4	…

⋮

改善結果



- 約30~40%の高速化に成功

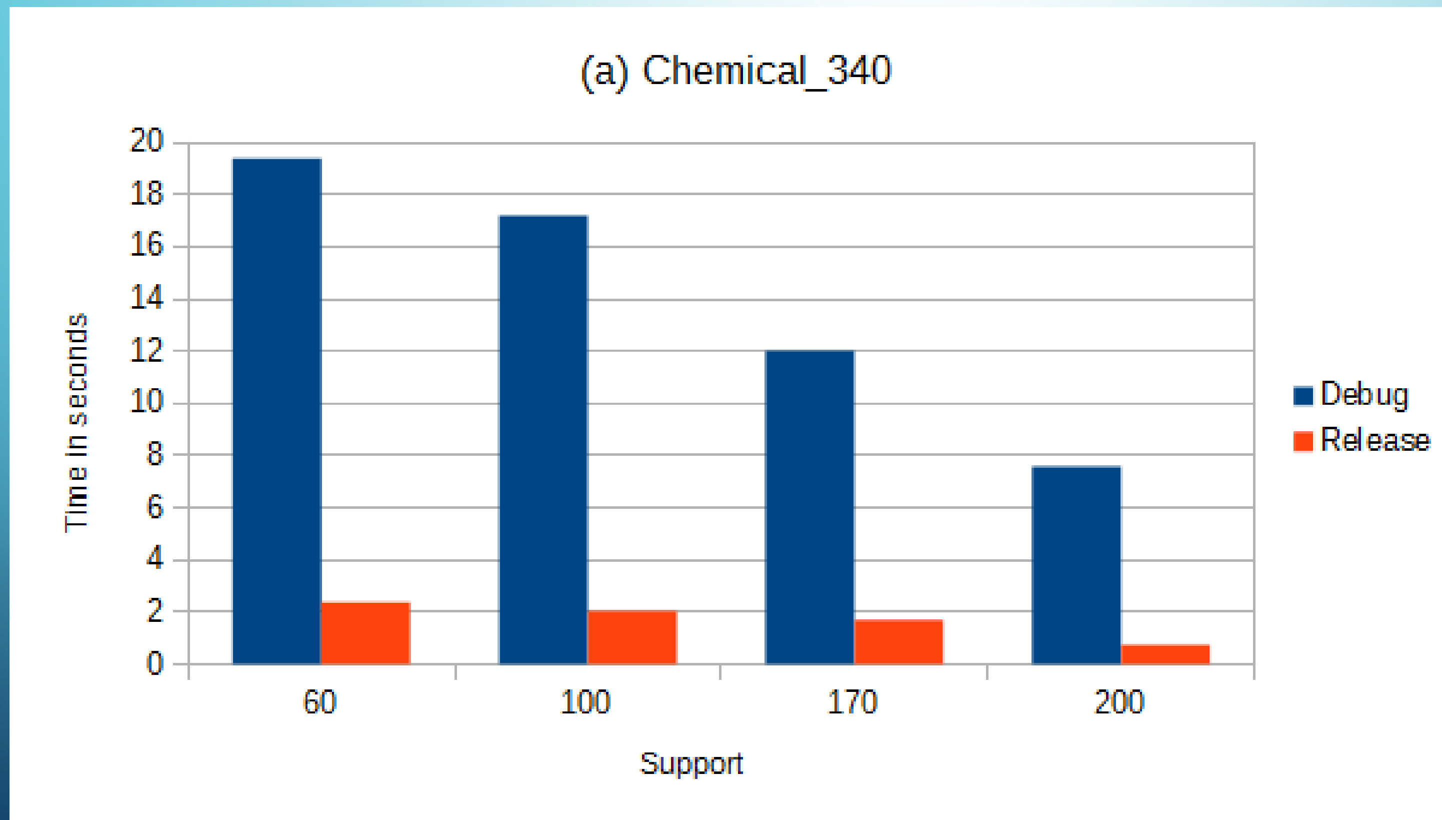
改善結果とその他改善案

- その他にも改善の余地がある
  - データ構造の見直し
  - 枝刈りの最適化
  - etc…

- プロファイリングを用いた改善

  ＝原因の解明と改善を繰り返すこと

改善結果 “Releaseビルド”



まとめ

- GPUで頻出部分グラフマイニングアルゴリズムの実装
- 世界の壁は厚かった
- プロファイリングによる原因の解明と改善策の発見

  →原因の一つは改善できた

今後の課題

- 今回改善できなかった部分の改善
  - データ構造の見直し
  - 枝刈りの最適化
  - 並列性の向上（起動するスレッド数の最適化）
- gSpanベースの他のアルゴリズムの高速化
  - 飽和パターン，ORIGAMI，etc…