グリッド・コンピューティング

08M37166 佐藤 賢斗

取り上げる論文

- Efficient Gather and Scatter Operations on Graphics Processors
  - Bingsheng He, Naga K. Govindaraju, Qiong Luo and Burton Smith
  - SC07

背景

- 汎用計算においてGPU(GPGPU)の利用
  - 高い並列性、メモリバンド幅
  - CUDAの登場
    - NVIDIAが提供するC言語の開発環境
  - E.g) Matrix multiplication, sorting, LU decomposition, FFT

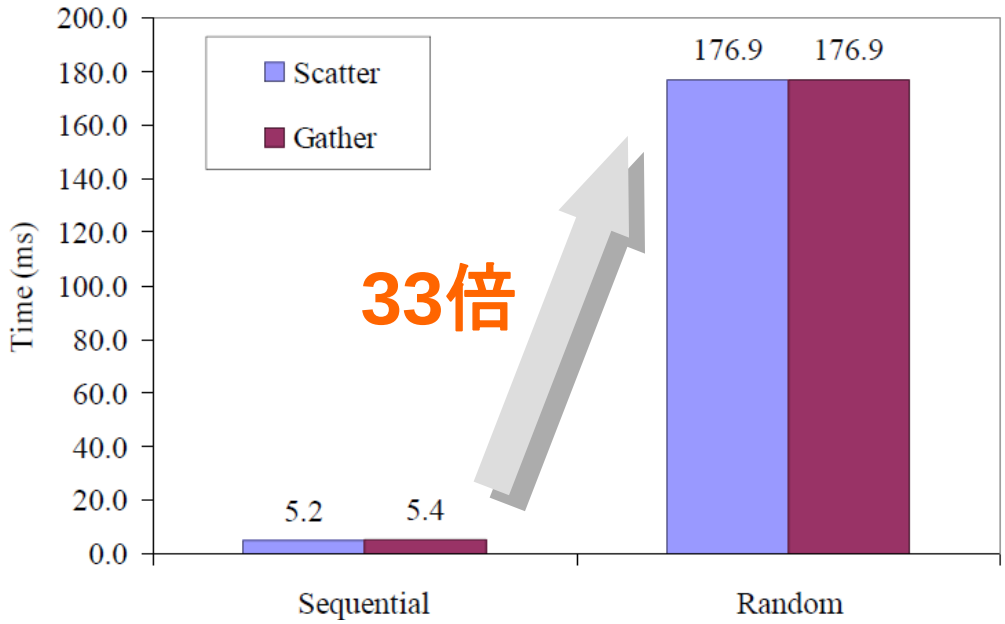
- Scatter & Gather 処理
  - e.g)
    - プロセス間通信: MPI
    - 並列プログラミング言語: ZPL, HPF
  - 最近のGPUではScatter & Gatherサポート
  - 多くの汎用計算で利用される

⇒ GPU上で汎用計算をする際、Scatter & Gatherの性能がアプリケーションに与える影響は大きい

問題点

- 既存の実装ではメモリバンド幅を有効に使用することができない
  - 特にRandom accessを伴うScatter & Gather

    ⇒キャッシュミスが多発



Input array size:
128MB

Random accessのスループット向上が重要

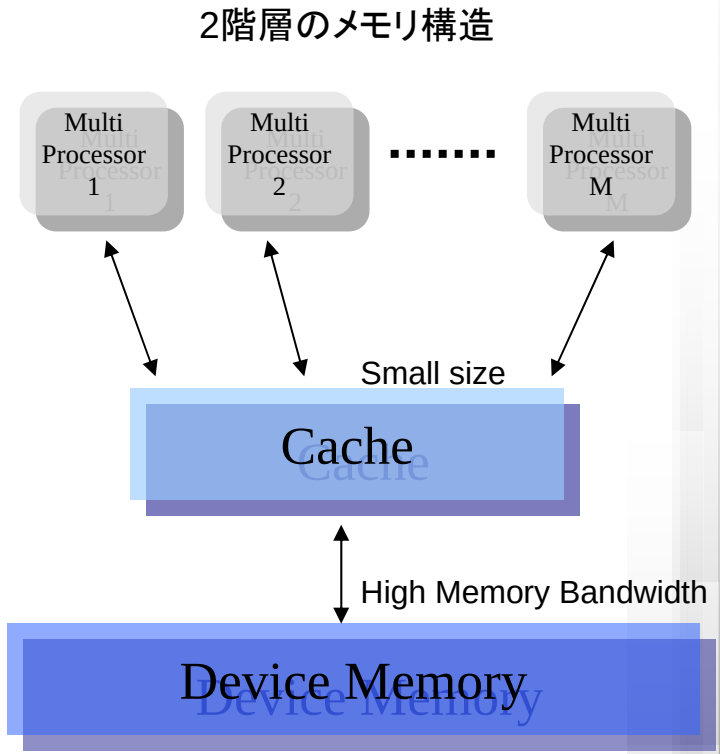
目的 & 成果

- 目的
  - GPUにおけるScatter & Gather 処理の最適化
    - Random accessのスループット向上による
- 成果
  - Scatter & Gather の時間をモデル化
  - Scatter & Gather の最適化
    - Multi-pass Scatter & Gatherを提案

⇒2-4倍の性能向上を実現

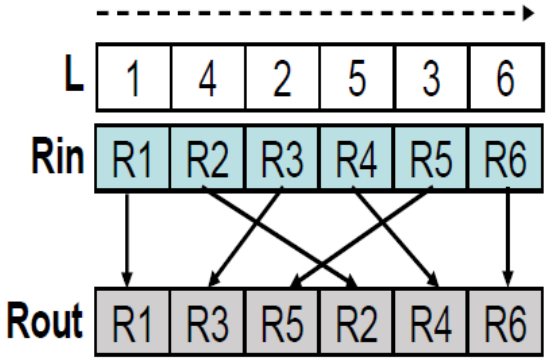
Memory Model on GPU

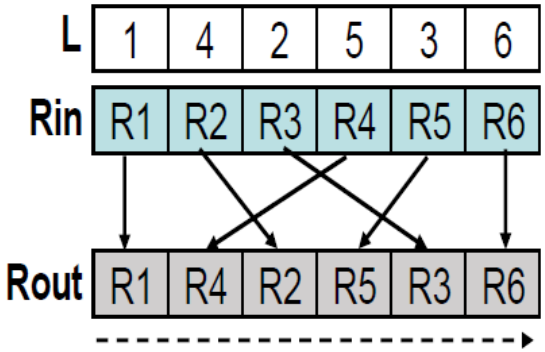
- GPUはM個のSIMD Multi processorから構成
  - 1つのmultiprocessor上では同じ命令が実行される
  - Thread: 最小の計算単位
    - 1 thread / processor
  - Thread warp:複数のThreadから構成される最小の命令単位
    - Warp内では同じ命令が実行
- メモリ階層：2階層
  - Cacheは全てのMulti processorで共有
- 各Multi processorはCacheへアクセス
  - Hit: Cache上にデータがある
    - Cacheからデータを取り出す
  - Miss: Cache上にデータがない
    - Device Memoryから対象のMemory blockを取り出す



Scatter & Gather の定義

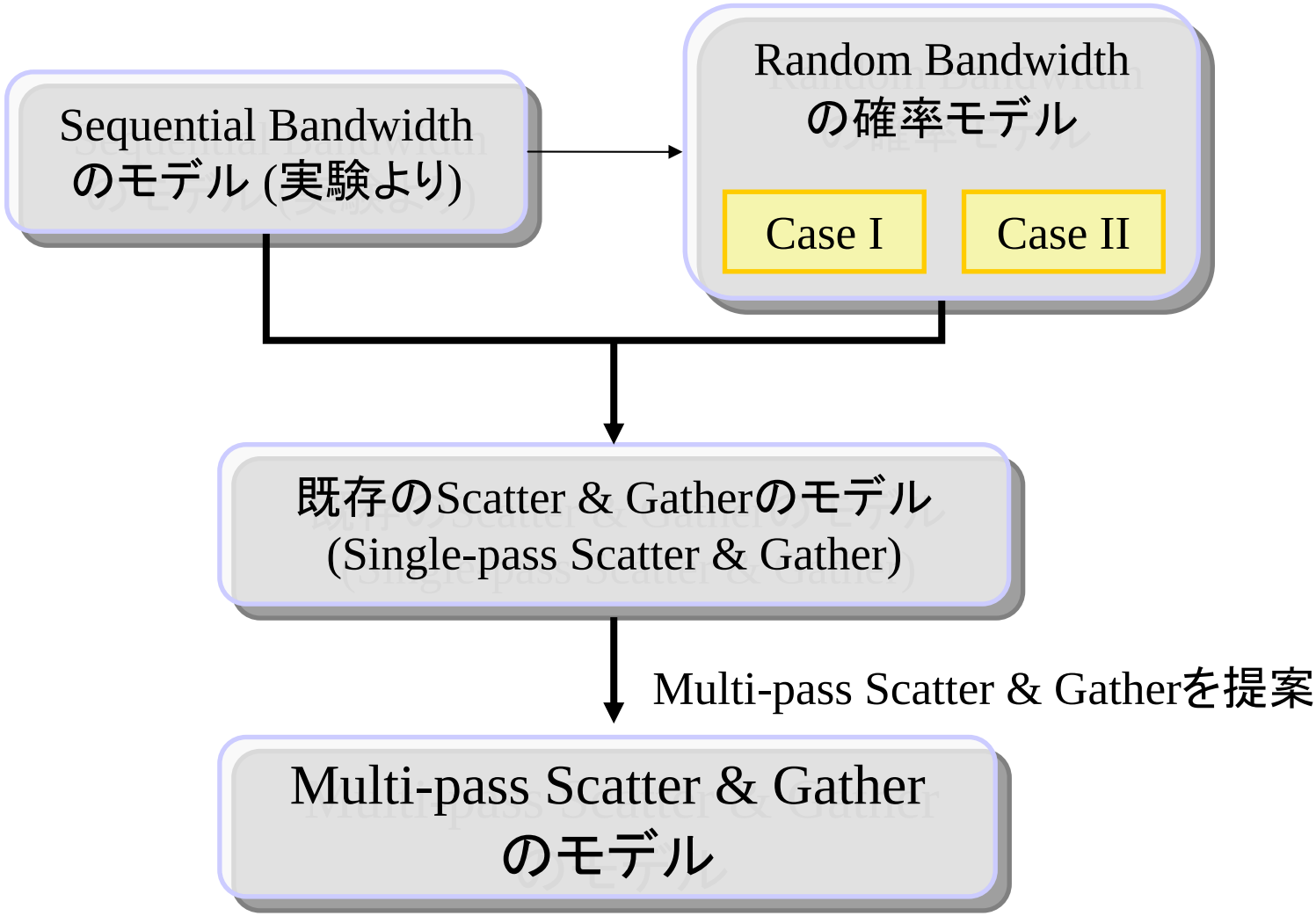
Primitive: **Scatter**	Primitive: **Gather**
Input: R_{in} [1, …, n], L[1, …, n].	Input: R_{in} [1, …, n], L[1, …, n].
Output: R_{out} [1, …, n].	Output: R_{out} [1, …, n].
Function: $R_{out}[L[i]]=R_{in}[i]$, i=1, …n.	Function: $R_{out}[i]=R_{in}[L[i]]$, i=1, …n.





- Scatter：連続領域のデータを指定した領域に書き込む
  - R_{in}: Sequential
  - L: Sequential
  - R_{out}: **Random**
- Gather：指定した領域のデータを連続領域に書き込む
  - R_{in}: **Random**
  - L: Sequential
  - R_{out}: Sequential

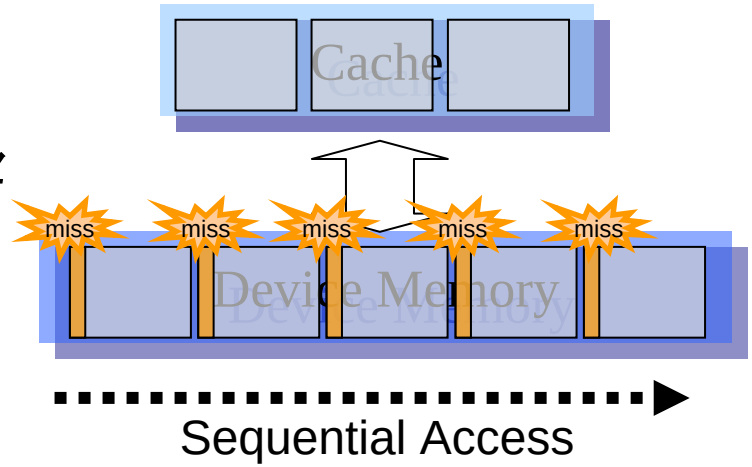
提案手法の流れ


Sequential Bandwidth
のモデル (実験より)
Random Bandwidth
の確率モデル
Case I
Case II
既存のScatter & Gatherのモデル
(Single-pass Scatter & Gather)
Multi-pass Scatter & Gatherを提案
Multi-pass Scatter & Gather
のモデル

The Sequential Bandwidth

- Sequential アクセス
  - アクセス領域を順にアクセス
  - 対象のMemory blockへ初めてアクセスするときに一度だけmissとなりあとはCacheから読み出される

  ⇒ (空間的)局所性が高いためスループットが高い
- 実際に配列をSequentialにアクセスする実験から算出
  - 配列サイズ : $2^i_{[MB]}$ (i=1,2, …, log D_m)
    - D_m : Device Memoryのサイズ
  - 重みつき平均により算出
- B_{seq} : Sequential bandwidth
  - t_i : $2^i_{[MB]}$ の配列へSequentialにアクセスするのに要した時間

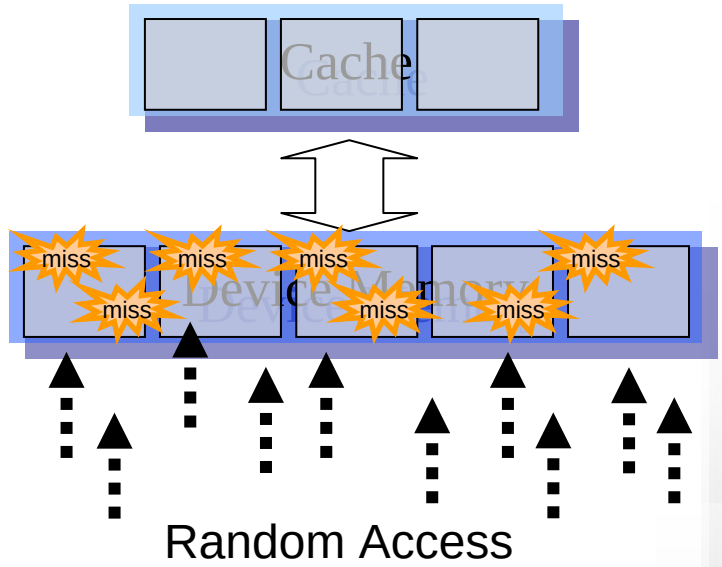


$$B_{seq} = \frac{1}{\sum_{i=1}^{n} 2^i} \cdot \sum_{i=1}^{n} \left(\frac{2^i}{t_i} \cdot 2^i \right)$$

The Random Bandwidth

- Random アクセス
  - 目的のデータが存在する領域を転々とアクセス
  - 一度アクセスしたMemory blockへ再びアクセスする可能性が大

  ⇒ 局所性が低いためSequential accessよりスループットが低くなる
- B_{seq} : Random bandwidth
  - z : 配列のタプルサイズ(bytes)
  - n : 配列のタプル数
  - t : 配列 (z, n)へのランダムアクセス時間
- t : 配列へのランダムアクセス時間
  - n : 配列のタプル数
  - h : Hit 率
  - z : 配列のタプルサイズ(bytes)
  - l : メモリブロックサイズ(bytes)
  - l' : キャッシュラインサイズ(bytes)
  - δ :1キャッシュラインの転送時間



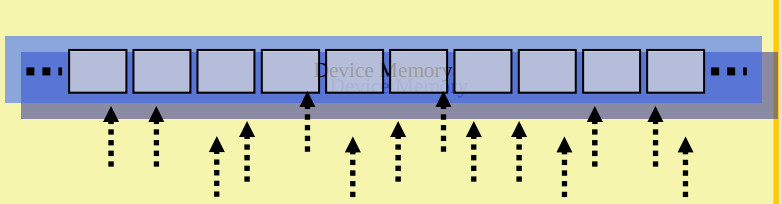
$$B_{rand} = \frac{n \cdot z}{t}$$

$$t = n \cdot \left((1-h) \cdot \frac{\lceil z/l \rceil \cdot l}{B_{seq}} + h \cdot \lceil z/l' \rceil \cdot \delta \right)$$

Hit 率のモデル化

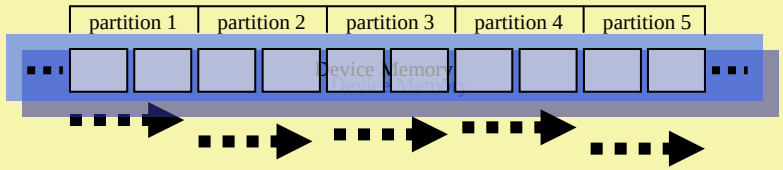
Random Bandwidthの確率モデル

Case I



一様なランダムアクセスの場合

Case II

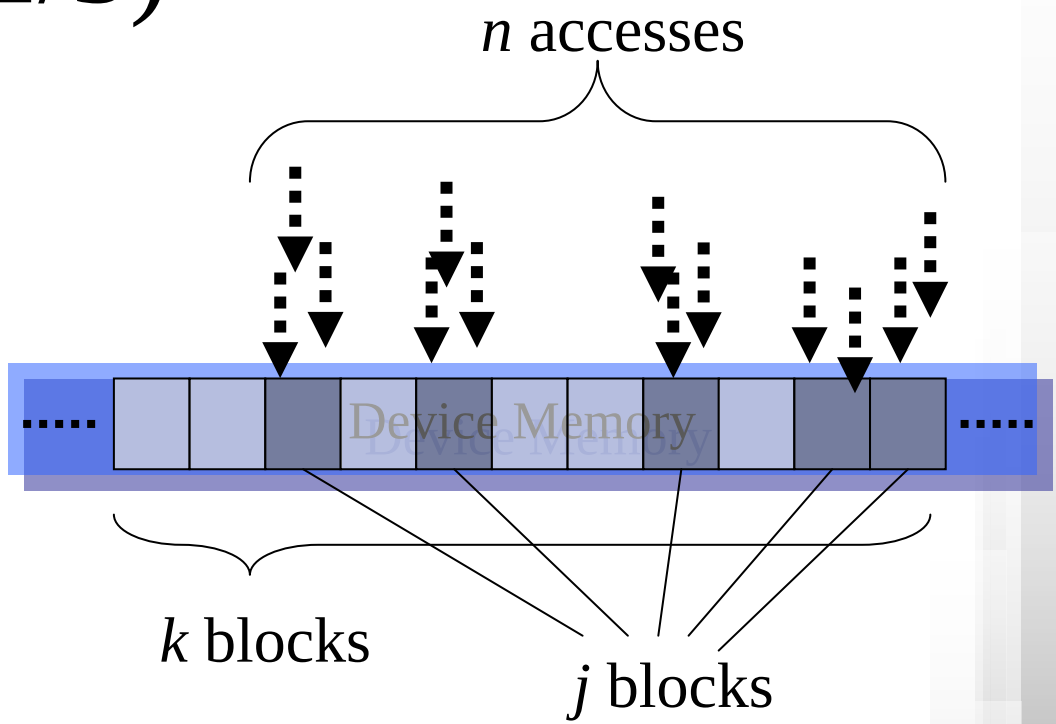


入力データのpartitionの数が既知の場合

Hit 率: Case I (1/3)

- E_j : n 回のアクセスのうち異なるj個のMemory blockにアクセスする場合の数
  - n : アクセス数
  - k : 入力配列を格納するのに必要なMemory block数
  - $S(X,Y)$: 第二種スターリング数
    - Y個のblockに対し1-Xのb番号を少なくとも一回以上用いて番号を振るときの場合の数

    ⇒ $S(n, j) \cdot j!$: n回のアクセスがj 個のblockに対してどの順にアクセスするか、の場合の数



$$E_j = {}_kC_j \cdot S(n, j) \cdot j!$$

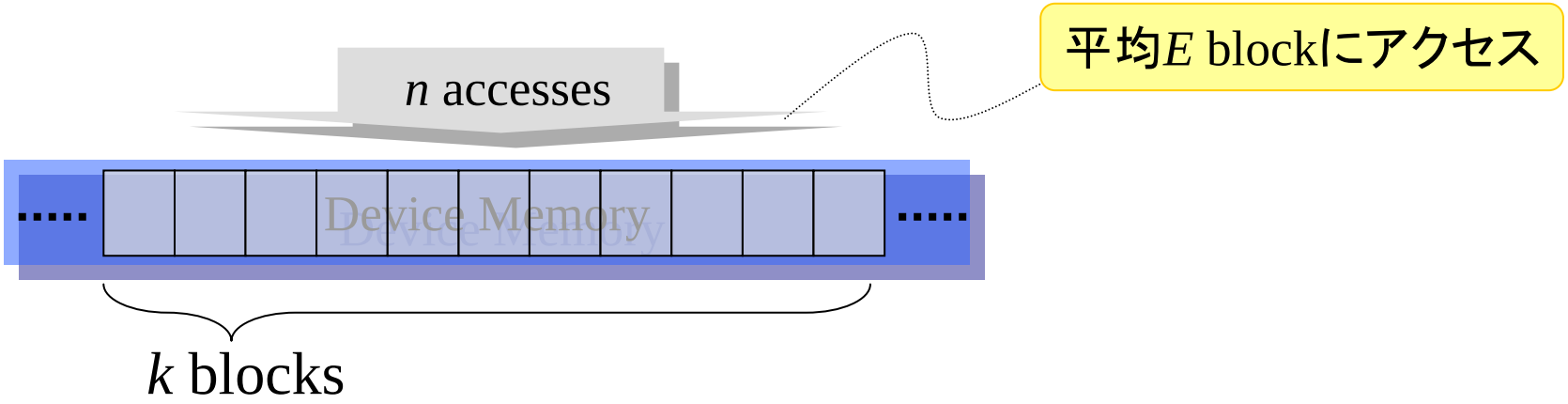
Hit 率: Case I (2/3)

- E_j : n 回のアクセスのうち異なるj個のMemory blockにアクセスする場合の数

$$E_j = {}_kC_j \cdot S(n, j) \cdot j!$$

- E : E_j の期待値 (平均)
  ⇒ n 回中、アクセスする平均のMemory block数

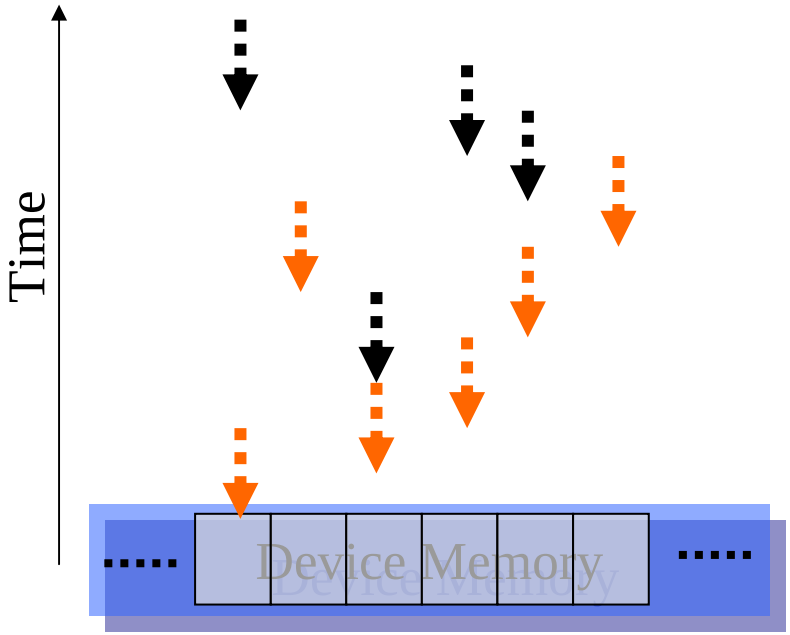
$$E = \sum_{j=1}^{n} \left(\frac{E_j}{k^n} \cdot j \right)$$



Hit 率: Case I (3/3)

- m : Miss率
  - E : n 回のアクセスが参照する平均のMemory block数
  - n : アクセス数
  - N : Cache line 数

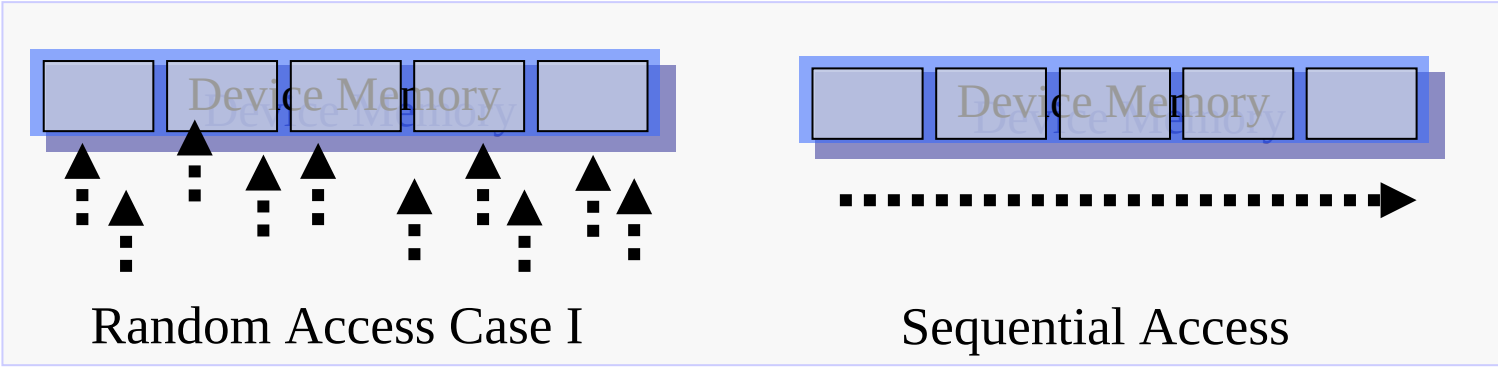
$$m = \begin{cases} \dfrac{E}{n} & (E \le N) \\ \dfrac{E + (\dfrac{E-N}{E}) \cdot (n-E)}{n} & otherwise \end{cases}$$



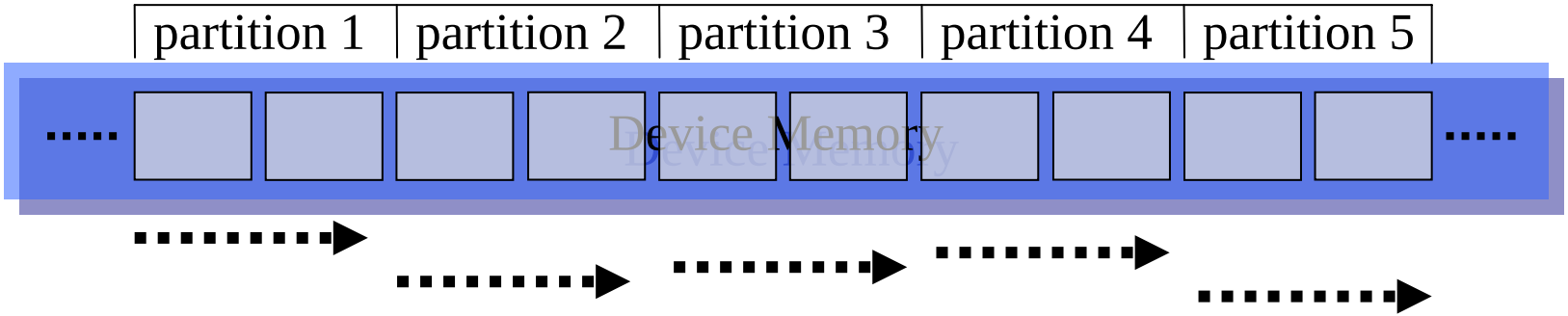
- $E <= N$: 一度アクセスしたblockは以後キャッシュ上に存在
  - Miss数 = E
  - Hit数= n - E
- $E > N$: 一度アクセスしたblockが以後キャッシュ上にのっているとは限らない
  - Miss数= E + (n - E) * ((E – N) / E)
  - Hit数 = (n - E) * (N / E)

$$h = 1 - m$$

Hit 率: Case II (1/2)



- Random Access Case II
  - partitionの数pが既知
  - 各partiionへのアクセス: Random
  - partition内のアクセス: Sequential



Hit 率: Case II (2/2)

- D_j : j個の異なるpartitionにアクセスする場合に数
  - p : partitionの数
  - w : 1warp中のthread数
    ⇒アクセスするtuple数
- D : 1warpがアクセスする平均のpartition数
- m : miss率
  - z : tuple size (bytes)
  - l : Memory block size (bytes)

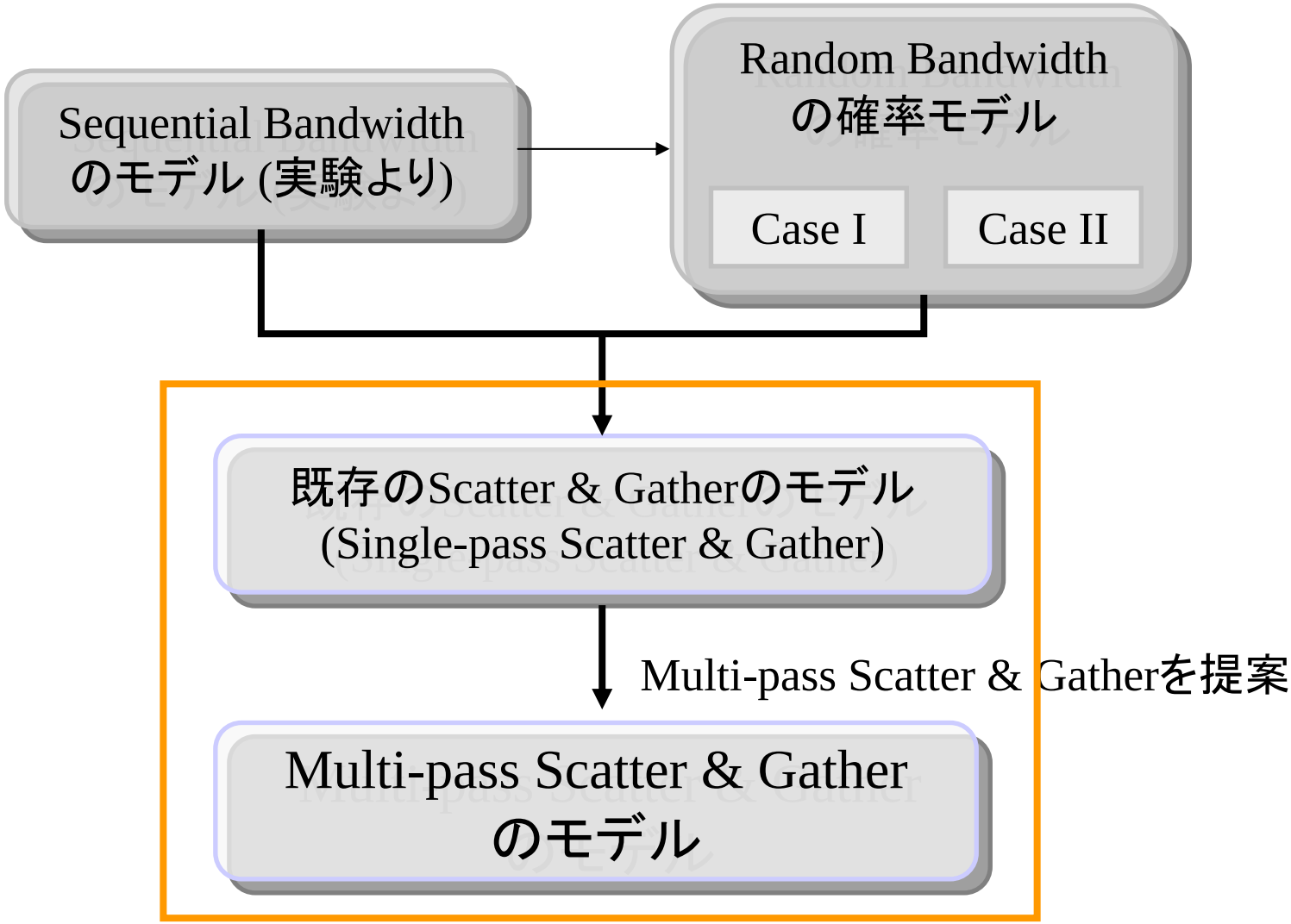
$$D_j = {}_pC_j \cdot S(w, j) \cdot j!$$

$$D = \frac{\sum_{j=1}^{\min(p,w)} \left(D_j, j\right)}{p^w}$$

$$m = \frac{D \cdot \left\lceil \frac{(w/D) \cdot z}{l} \right\rceil}{w}$$

$$h = 1 - m$$

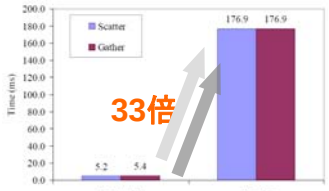
提案手法の流れ


Sequential Bandwidth
のモデル (実験より)
Random Bandwidth
の確率モデル
Case I
Case II
既存のScatter & Gatherのモデル
(Single-pass Scatter & Gather)
Multi-pass Scatter & Gatherを提案
Multi-pass Scatter & Gather
のモデル

既存のScatter & Gatherのモデル式

- Scatter : 連続領域のデータを指定した領域に書き込む
  - R_{in}: Sequential
  - L: Sequential
  - R_{out}: **Random**
- Gather : 指定した領域のデータを連続領域に書き込む
  - R_{in}: **Random**
  - L: Sequential
  - R_{out}: Sequential

- $T_{scatter}$, T_{gather} : Scatter, Gather時間
  - $\|X\|$: 配列Xのサイズ(bytes)



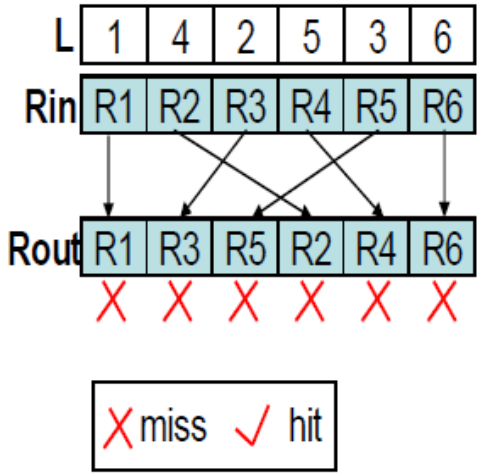
$$T_{scatter} = \frac{\|R_{in}\|}{B_{seq}} + \frac{\|L\|}{B_{seq}} + \frac{\|R_{out}\|}{B_{rand}}$$

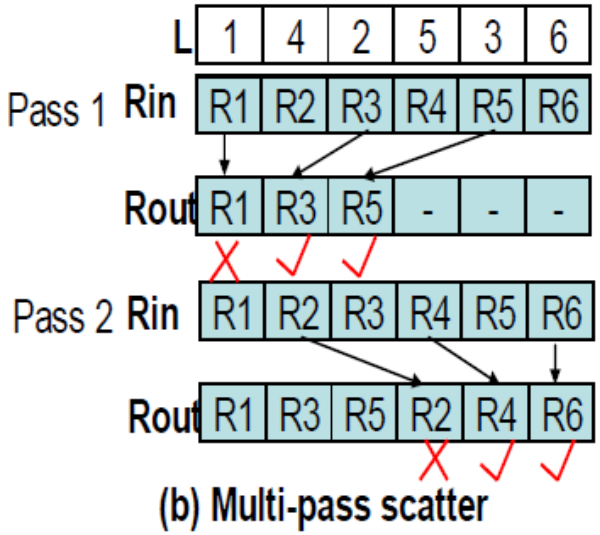
$$T_{gather} = \frac{\|R_{in}\|}{B_{rand}} + \frac{\|L\|}{B_{seq}} + \frac{\|R_{out}\|}{B_{seq}}$$

⇒ 単純な実験からB_{rand} はB_{seq} の約1/33のスループットであるため、最適化が必要

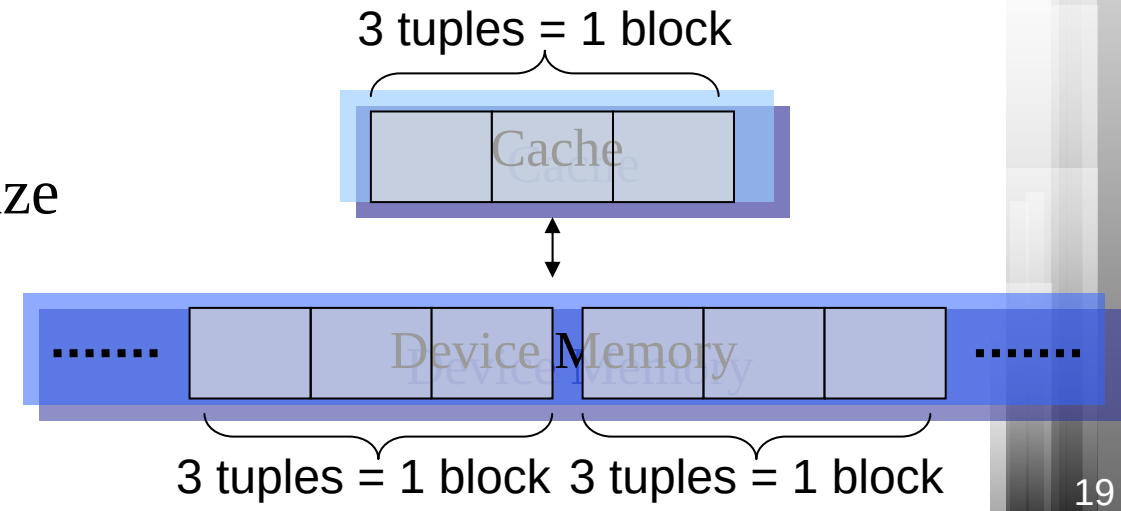
最適化手法: Multi-pass Scatter

- (a) Single-pass scatter
  - 既存手法
- (b) Multi-pass scatter
  - 提案手法



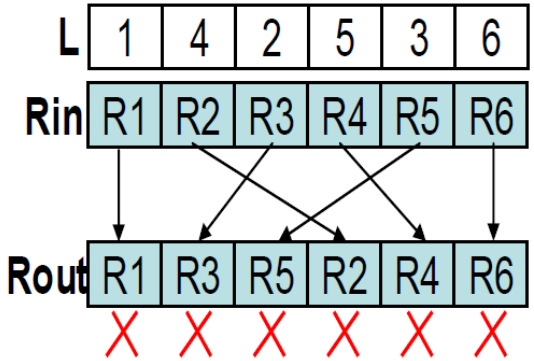


- 例)
  - Cache size
    = 1 memory block size
    = 3 tuples



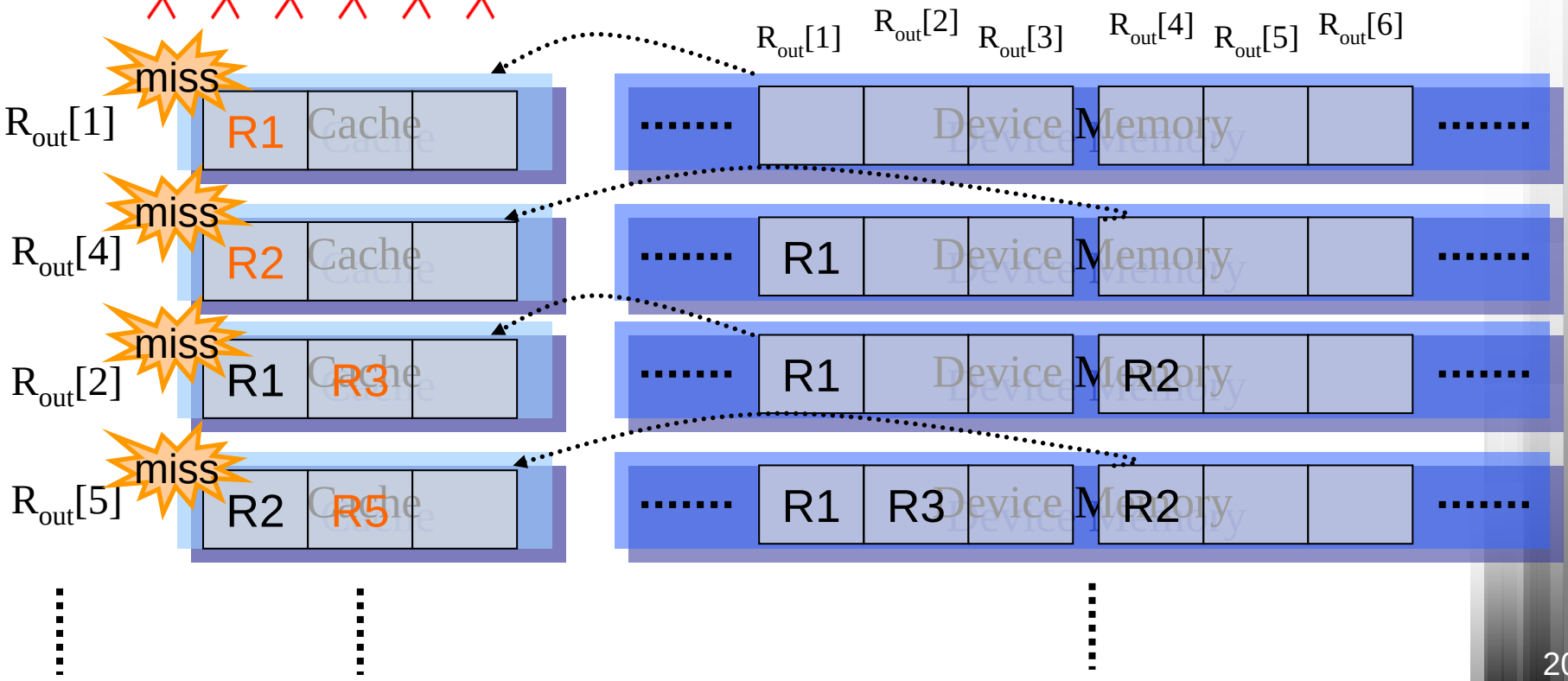
※Multi-pass Gatherも同様な議論のためScatterのみ紹介

Single-pass Scatterの問題点

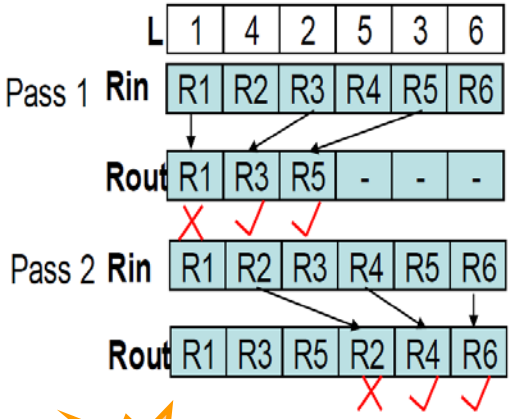


- $L[i]$ が一様にランダムな場合
  - キャッシュミスが多発
    - B_{rand} が低下

⇒ Scatter のスループット低下

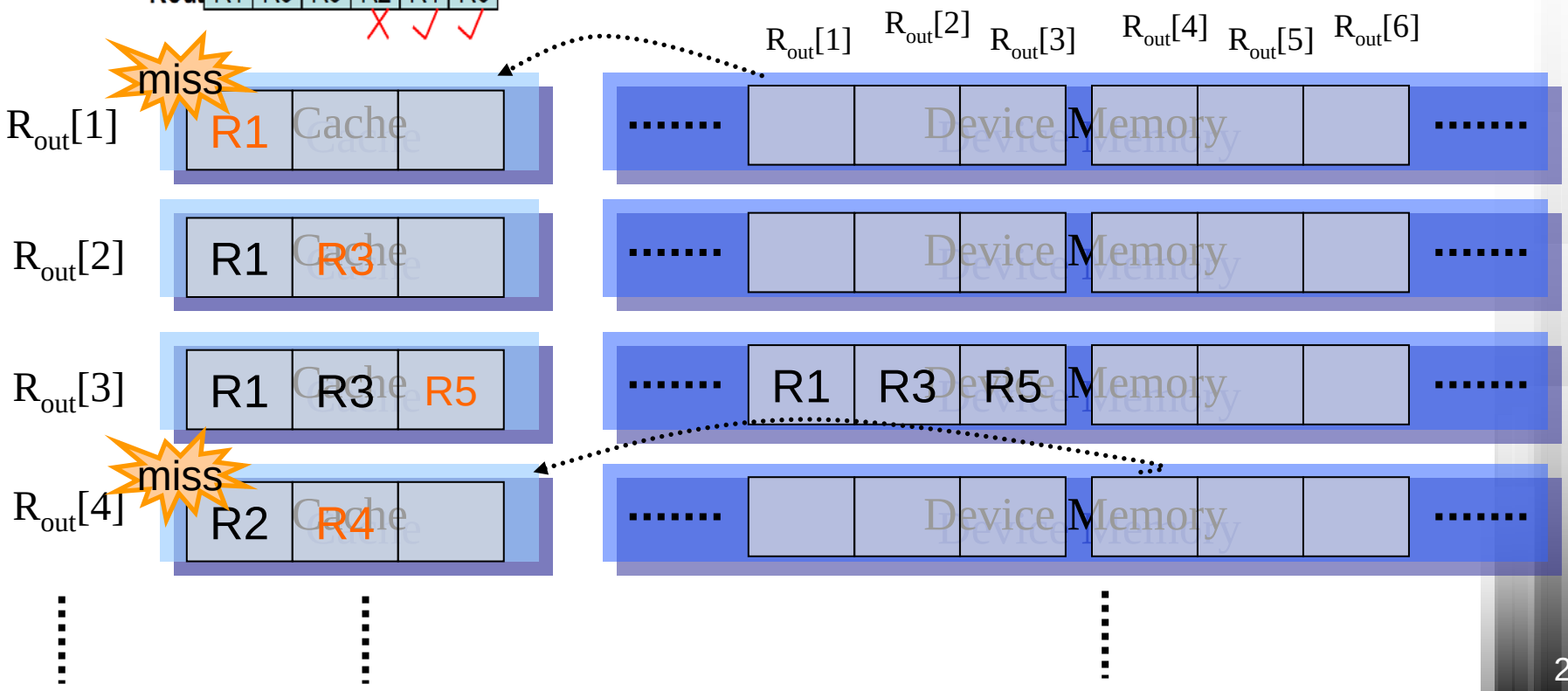


Multi-pass Scatterについて

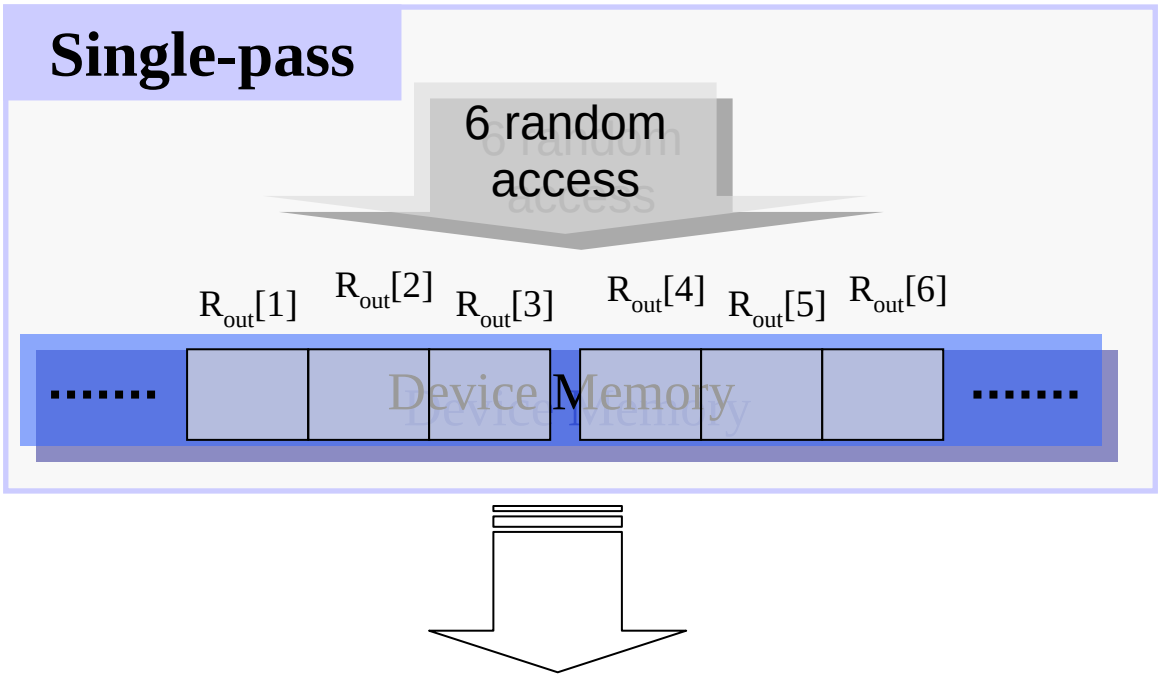


- 複数回に分けてScatterを行う
  - $R_{out}[i]$ へのアクセスに局所性を持たせる
    - B_{rand} の低下を抑制

⇒ scatterのスループット向上



Single-pass & Multi-pass

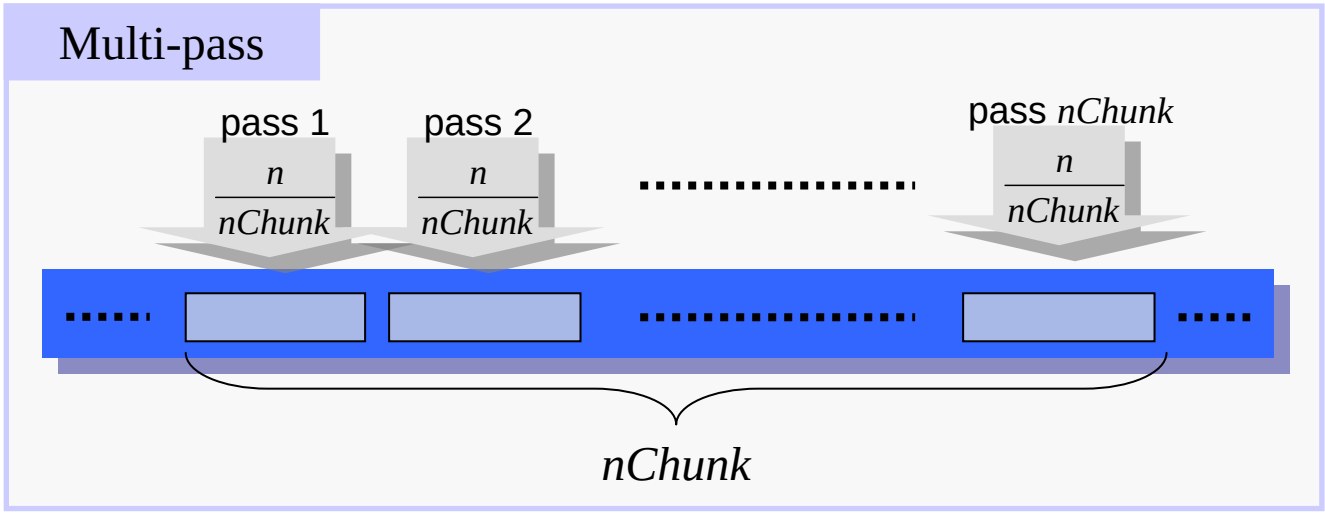


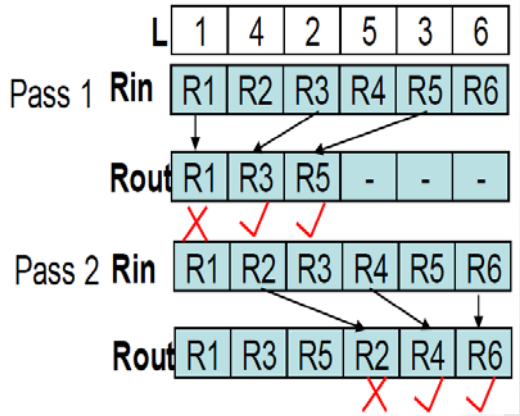
- 6回のアクセスが $R_{out}[1]$ - $R_{out}[6]$に及ぶ

- アクセスを分割
  - 前半3回のアクセス
    - $R_{out}[1]$ - $R_{out}[3]$
  - 後半3回のアクセス
    - $R_{out}[4]$ - $R_{out}[6]$
- 一方でSequentialアクセスの時間が増加

⇒ 最適なpass数を決定する必要がある

Multi-pass Scatterのモデル式





- $T'_{scatter}$: Multi-pass scatterの時間
  - *nChunk* : partition数
    ⇒ *pass数*
  - B'_{rand} : B_{rand}において
    - *n* ⇒ *n/nChunk*,
    - *k* ⇒ 1 partitionの Memory block数

$$T'_{scatter} = \frac{\left(\|R_{in}\| + \|L\|\right) \cdot nChunk}{B_{seq}} + \frac{\|R_{out}\|}{B'_{rand}}$$

↑このモデル式を用いて
最適な*nChunk*数 (pass数)を求める

評価環境

- GPU-Machine
  - GPU: G80 (Geforce 8800 GTX)
  - CPU: Intel core2 Quad
  - OS: Windows XP
- CPU-Machine
  - CPU: AMD Opteron 280 dual-core * 2
  - OS: Windows Server 2003

詳細情報

	GPU(G80)	*CPU(Intel)*	*CPU(AMD)*
Processors	1350MHz × 8 × 16	2.4 GHz × 4 (Quad-core)	2.4GHz × 2 × 2 (two dual-core)
Cache size	392 KB	L1: 32KB × 4, L2: 8MB	L1: 64KB × 2 × 2, L2: 1MB × 2 × 2
Cache block size (bytes)	256	L1: 64, L2: 128	L1: 128, L2: 128
Cache access time (cycle)	10	L1: 3, L2: 11	L1: 3, L2: 9
DRAM (MB)	768	1024	16384
DRAM latency (cycle)	200	138	138
Bus width (bit)	384	64	64
Memory clock (GHz)	1.8	1.3	1.0

評価項目

- モデルの精度の検証
  - Sequential single-pass scatter & gather
  - Random single-pass scatter & gather
    - Case I, Case II

  ※ CPU-GPU間の転送時間は含まない

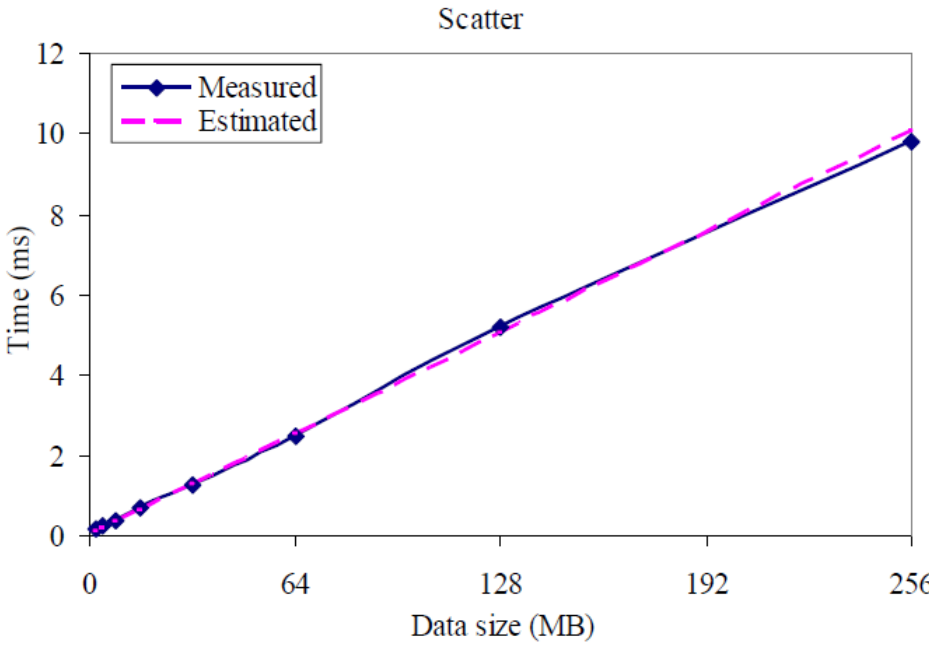
- Single/Multi-pass scatterの比較

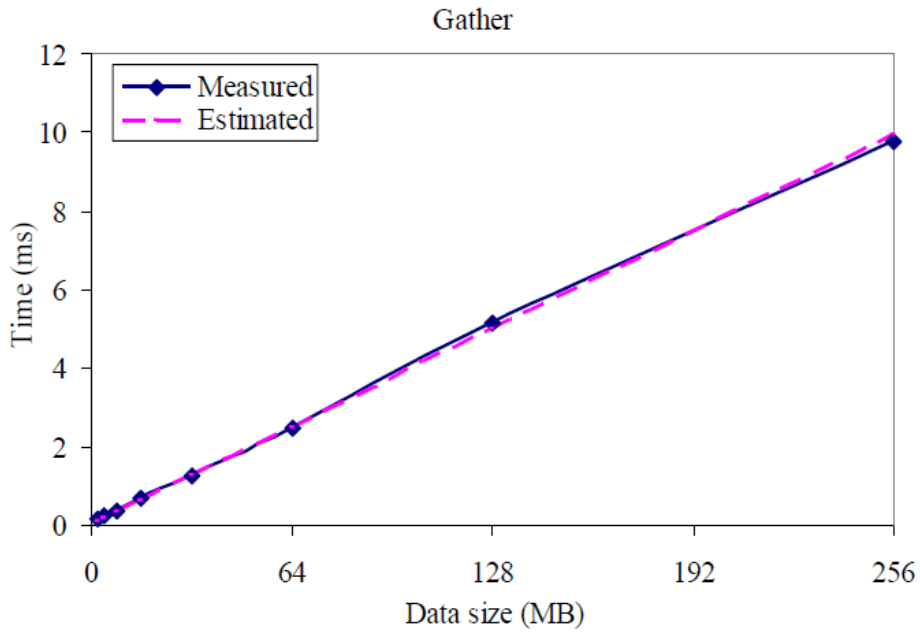
  ※ CPU-GPU間の転送時間は含まない

- Multi-pass Scatter & Gatherを用いたアプリの性能評価
  - Radix sort (MSD基数ソート)
  - Hash search
  - Sparse-matrix vector multiplication (SpMV)

  ※ CPU-GPU間の転送時間は含む

Sequential Single-pass Scatter & Gather





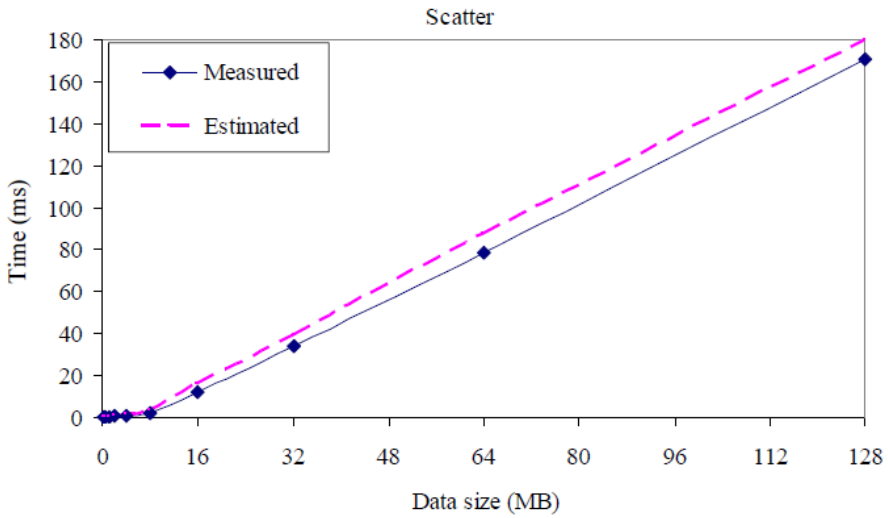
- tuple size: 8 bytes
- 87%以上の精度
  - v : Measured
  - v' : Estimated $1 - \frac{|v - v'|}{v}$

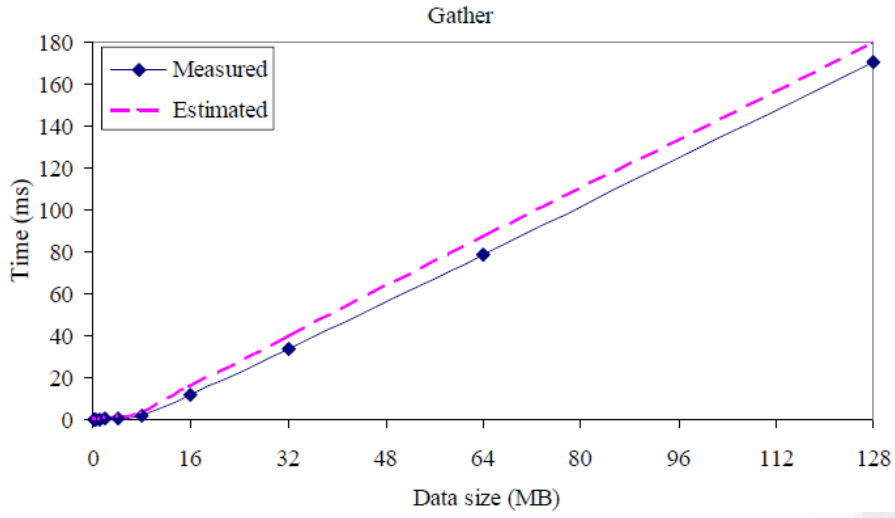
$$T_{scatter} = \frac{\|R_{in}\|}{B_{seq}} + \frac{\|L\|}{B_{seq}} + \frac{\|R_{out}\|}{B_{seq}}$$

$$T_{gather} = \frac{\|R_{in}\|}{B_{seq}} + \frac{\|L\|}{B_{seq}} + \frac{\|R_{out}\|}{B_{seq}}$$

※CPU – GPU間の転送時間は含まれていない

Random Single-pass Scatter & Gather: Case I





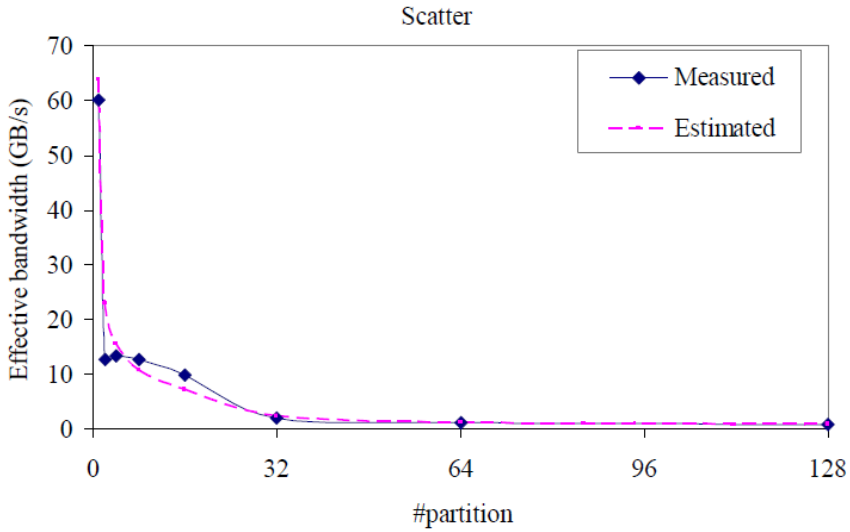
- tuple size: 8 bytes
- Array sizeが小さいときはキャッシュに載りやすくなる
- 90%以上の精度

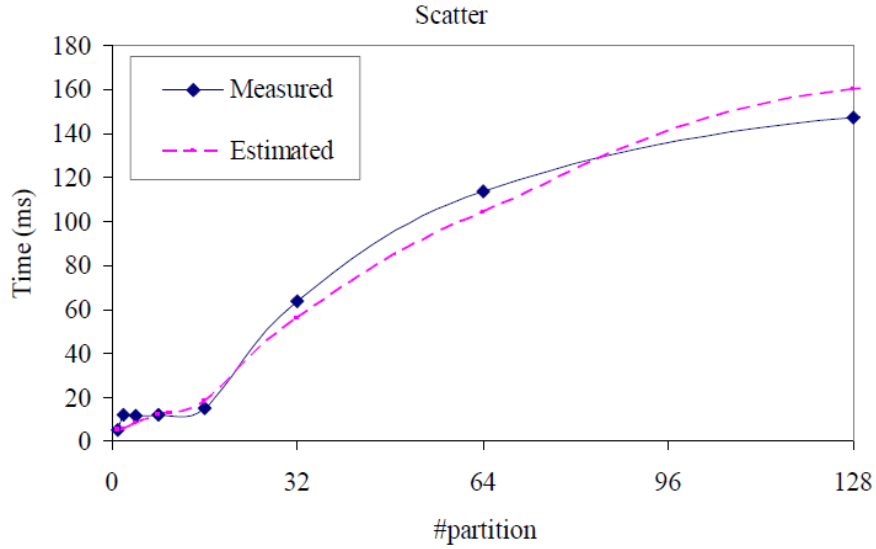
$$T_{scatter} = \frac{\|R_{in}\|}{B_{seq}} + \frac{\|L\|}{B_{seq}} + \frac{\|R_{out}\|}{B_{rand}}$$

$$T_{gather} = \frac{\|R_{in}\|}{B_{rand}} + \frac{\|L\|}{B_{seq}} + \frac{\|R_{out}\|}{B_{seq}}$$

※CPU – GPU間の転送時間は含まれていない

Random Single-pass Scatter & Gather: Case II





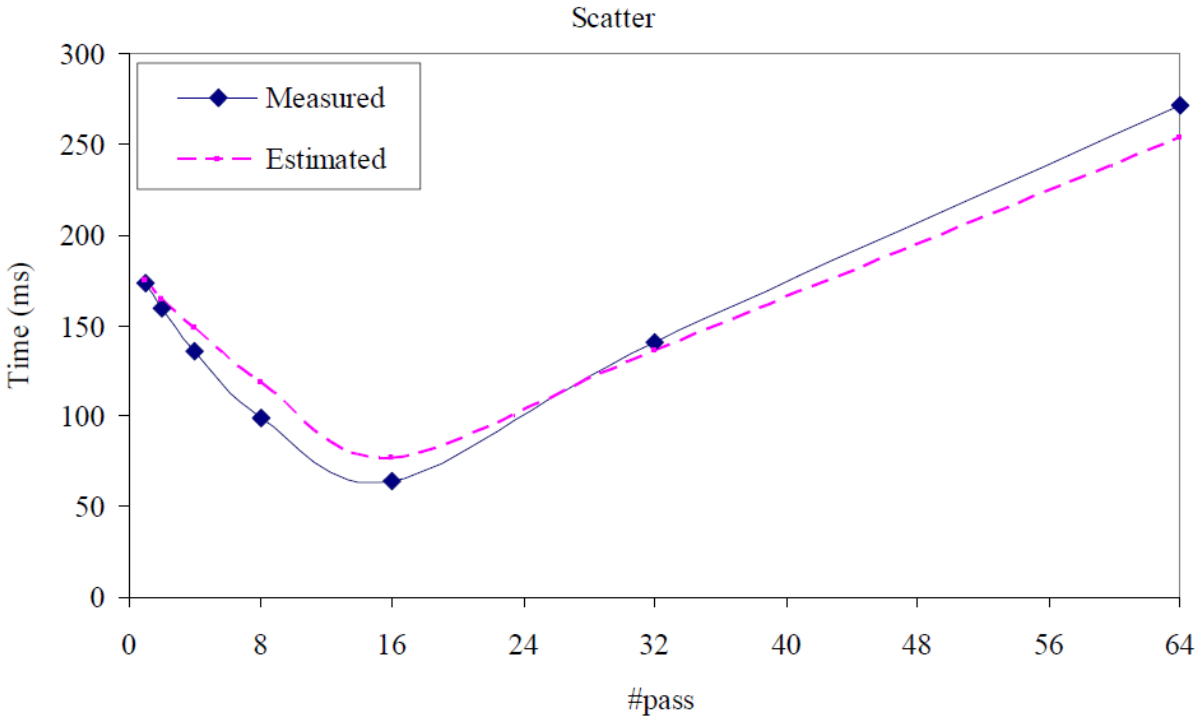
- Array size: 128MB
  - tuple size: 8 bytes
  - ランダムにPartition IDを割り当てる
- #partitions=32でスループット低下
  - partition > thread warp size (32 thread)
  - 局所性が低下
- 82%以上の精度

$$T_{scatter} = \frac{\|R_{in}\|}{B_{seq}} + \frac{\|L\|}{B_{seq}} + \frac{\|R_{out}\|}{B_{rand}}$$

$$T_{gather} = \frac{\|R_{in}\|}{B_{rand}} + \frac{\|L\|}{B_{seq}} + \frac{\|R_{out}\|}{B_{seq}}$$

※CPU – GPU間の転送時間は含まれていない

Multi-pass Scatter

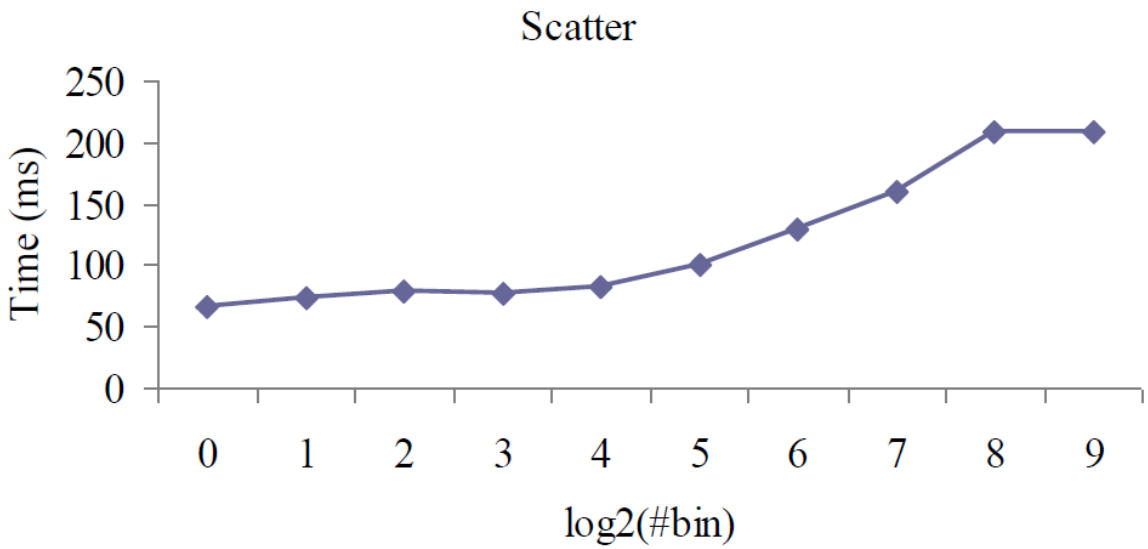


- Array size: 128MB
  - tuple size : 8bytes
- #pass=16が最適値
  - < 16 : 局所性の低下
  - > 16 : Sequentialアクセスが Overhead
- 平均86%の精度

$$T'_{scatter} = \frac{\left(\|R_{in}\| + \|L\|\right) \cdot nChunk}{B_{seq}} + \frac{\|R_{out}\|}{B'_{rand}}$$

※CPU – GPU間の転送時間は含まれていない

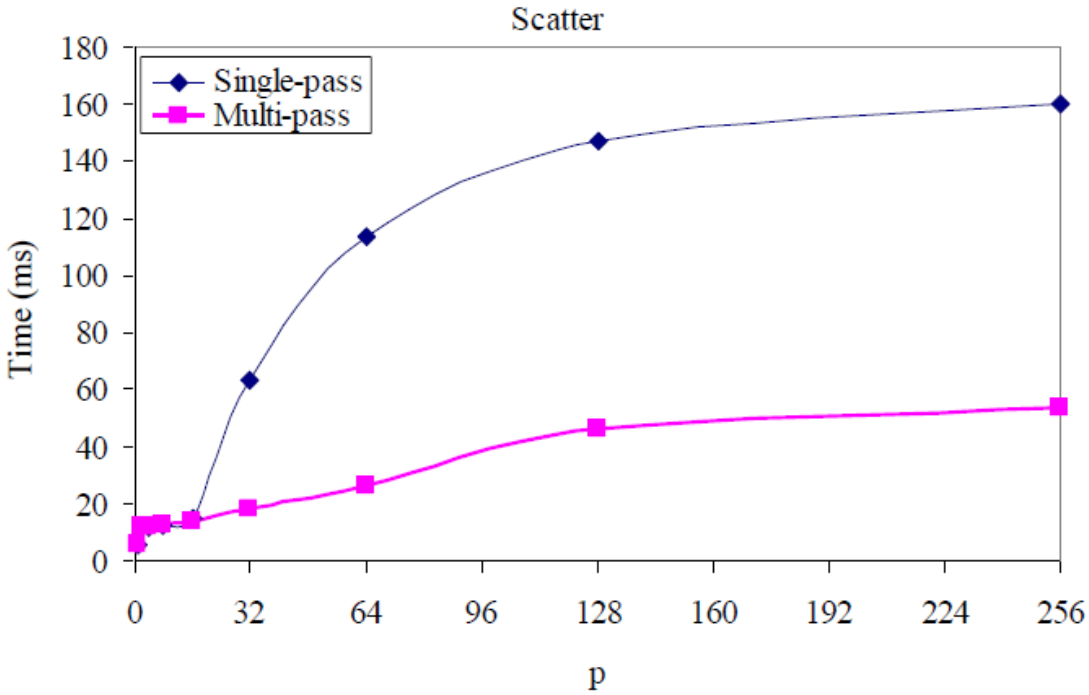
配列をSegmentに分割したときの



- Array size : 128MB
  - 配列をSegmentに分割しそれぞれに対してScatterを実行したときの時間

- Segment数が増えるにつれて時間が増加
  - segmentごとの局所性が悪化するため
- #bin=9 (segment size = 256kB)でフラット
  - cache size > 256kBであるため

Single-pass VS. Multi-pass Scatter



- Array size : 128MB
  - tuple size : 8 bytes

- p <= 8
  - 最適pass数=1のためSingle-passと同じ時間
- P > 8
  - モデルより最適なpass数(>1)を決定することにより最大で約3倍の性能向上

$$T'_{scatter} = \frac{(\|R_{in}\| + \|L\|) \cdot nChunk}{B_{seq}} + \frac{\|R_{out}\|}{B'_{rand}}$$

Radix sort：基数ソート (1/2)

キー(桁の値)に着目してソート

170, 45, 75, 90, 2, 24, 802, 66

という数列を1の位についてソートすると、

170, 90, 2, 802, 24, 45, 75, 66

となる。さらに、10の位についてソートすると、

2, 802, 24, 45, 66, 170, 75, 90

となる。最後に、100の位についてソートすると、

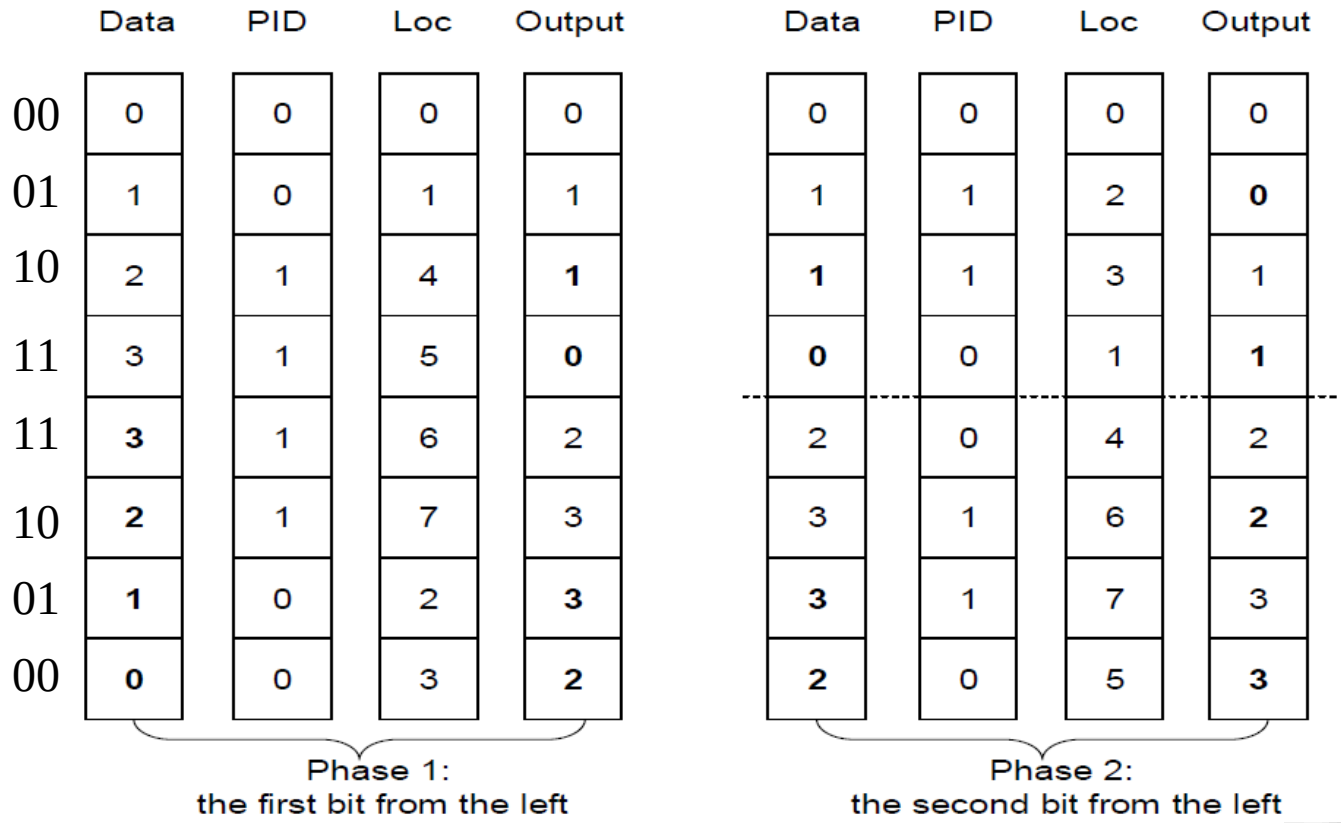
2, 24, 45, 66, 75, 90, 170, 802

Wikipediaより

Radix sort : 基数ソート (2/2)

0 1
A1[4 4]
分布表

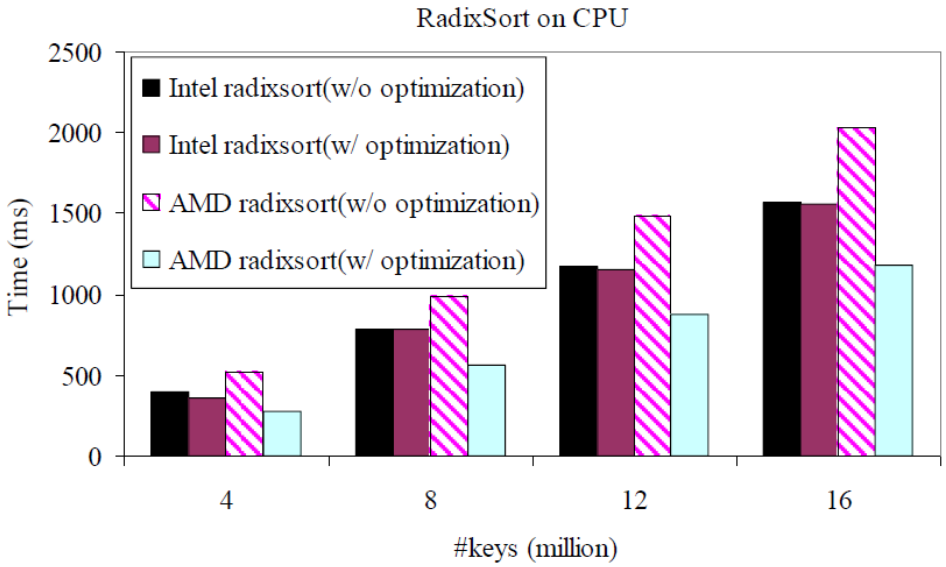
0 1
A2[4 8]
ヒストグラム

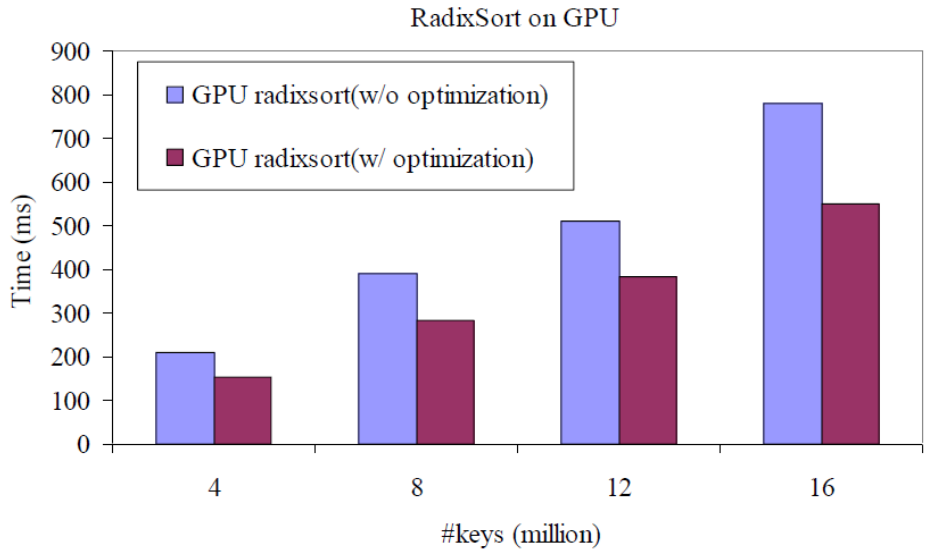


- 対象bit = n
  - Loc[A2[n] – A1[n]] ~ Loc[A2[n] – 1]にScatter

n=0 ⇒ 0 3

Radix sort on CPU & GPU





- Multi-pass Scatter によりCPU & GPU共に性能向上

まとめ

- メモリバンド幅をモデル化
  - Sequential, Random access
- バンド幅を有効に利用するScatter & Gatherを提案
  - Multi-pass Scatter & Gather

  ⇒ 最適なpass数をモデルから計算
- 汎用計算の性能向上を実現
  - Radix sort, Hash search, Sparse-matrix vector multiplication

今後の課題

- その他の汎用計算への適用
  - e.g.) LU decomposition, FFT
- 提案 Multi-pass scatter & gatherのIBM Cellへの適用
- 共有メモリを持たないプログラミングモデルへの適用
  - e.g) クラスタ環境、MPP (Massively Parallel Processor)