中核機関報告：筑波大学

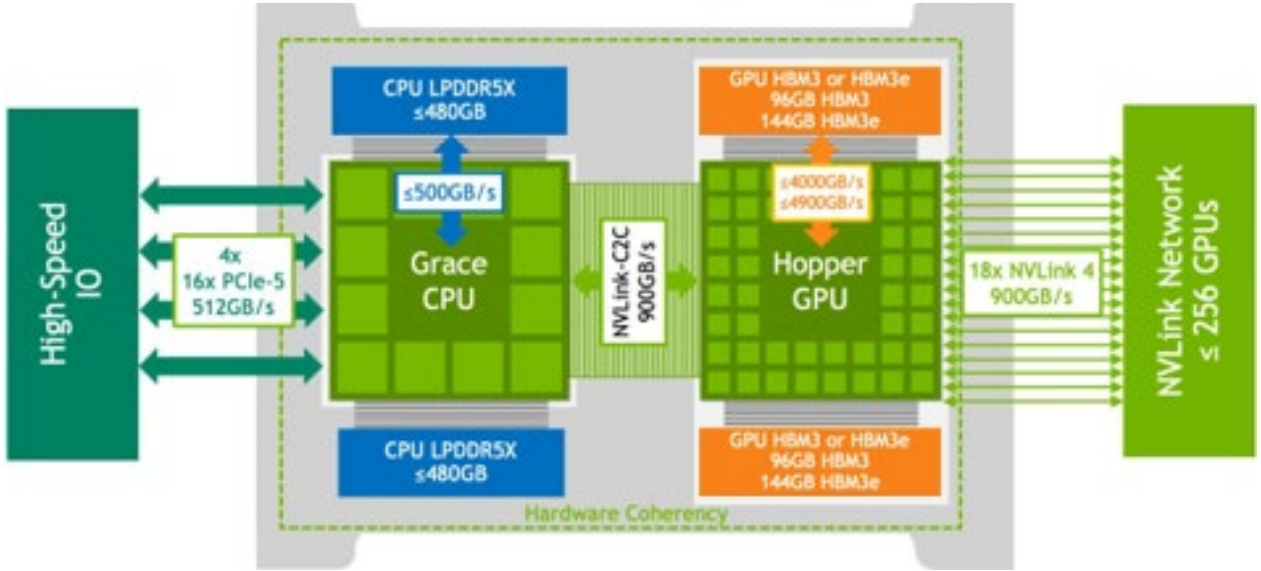
額田　彰
筑波大学計算科学研究センター

本日の内容

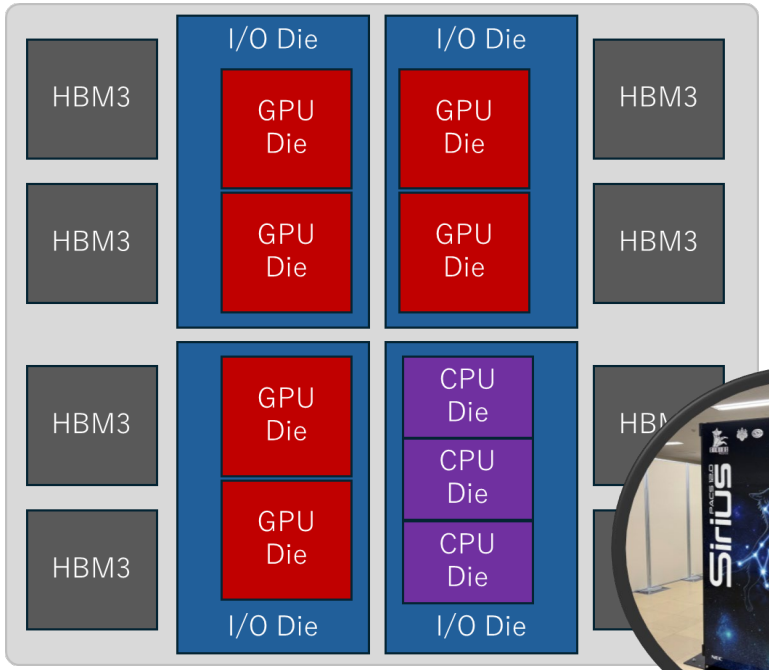
- Unified Memoryの特性分析
- OpenACCのカーネル最適化事例紹介
- AMD GPUの性能評価
- GPUプログラミング教材関連

Unified MemoryとManaged Memory

- ともにCPUとGPUの両方からアクセスできるメモリ
  - Unified memoryは計算環境の対応が必須
  - GPUプログラミングを容易にする機能



NVIDIA GH200
JCAHPC Miyabi-G

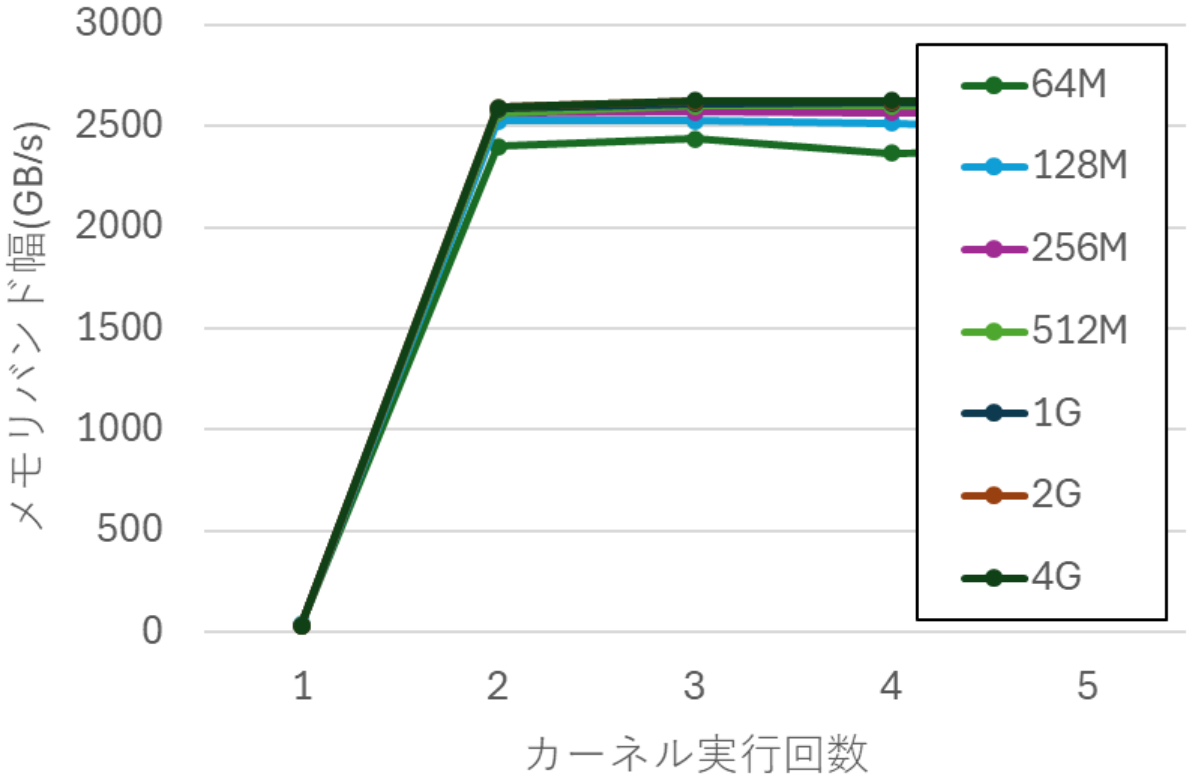


AMD MI300A
筑波大CCS Sirius

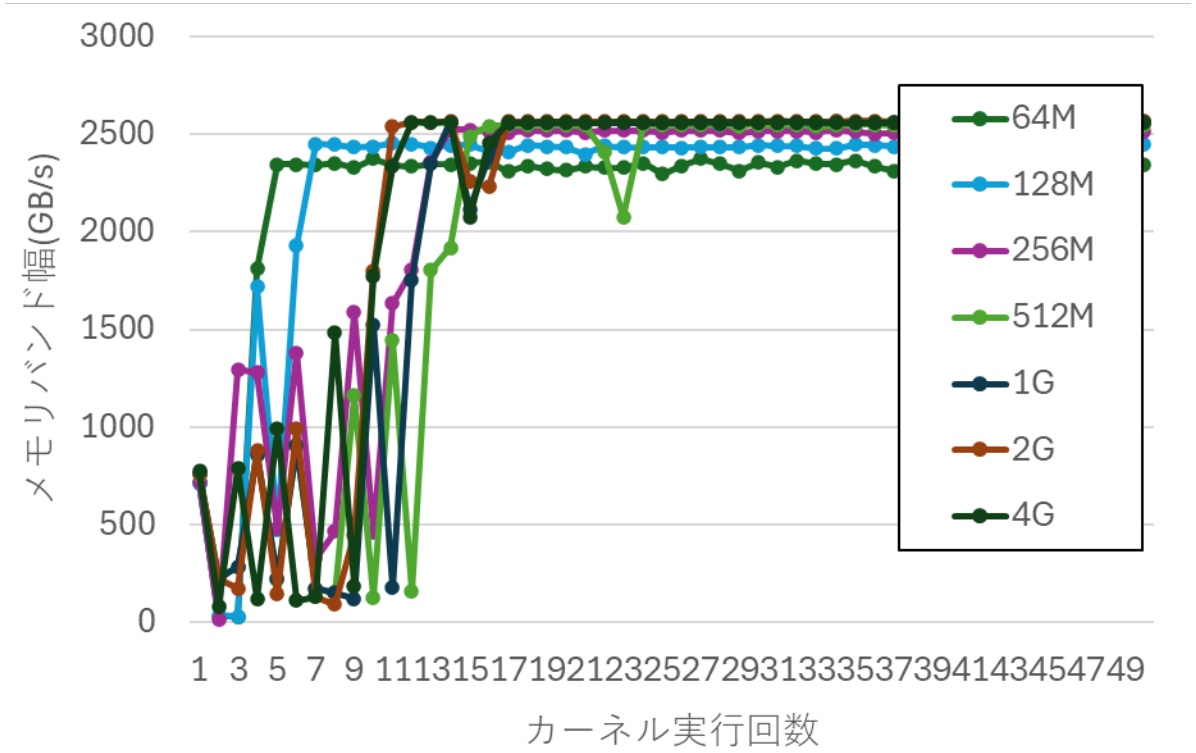
先週運用開始

両者の挙動の違い

CPUで初期化した後、GPUカーネルがアクセスを繰り返す場合
（配列要素に１を足すだけの処理）

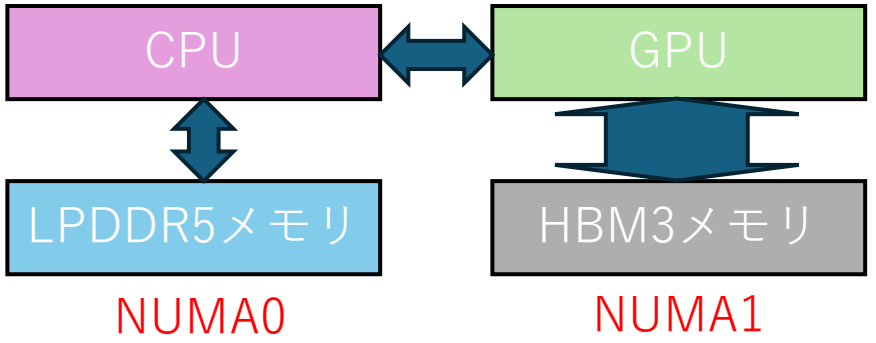


Managed Memoryの場合
最初のカーネル実行でGPUメモリに全て移動



Unified Memoryの場合
何回かのカーネル実行でGPUメモリに移動が完了

Unified memoryのページの状態



NVIDIA GH200の構成

libnumaのmove_pages()関数を使用

move_pages(0, 1, &ptr, NULL, &res, 0);

ptrに示すページがどのNUMAノードにあるかをresに返す

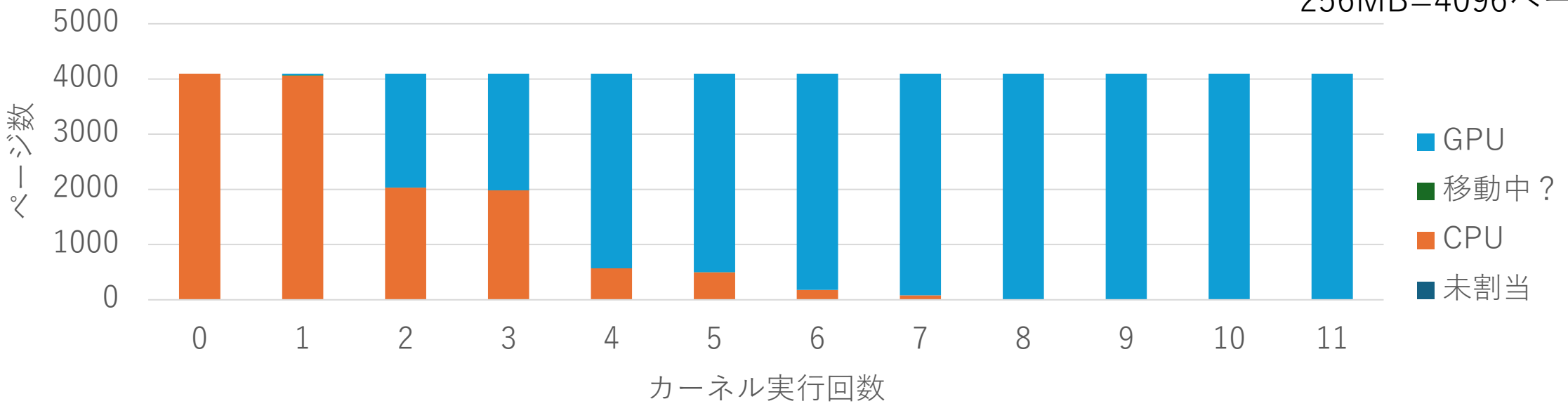
move_pages()の返り値	resの値	ページの状態
-1		物理メモリが未確保
0	0	CPUメモリにある
0	<0	移動中？
0	1	GPUメモリにある

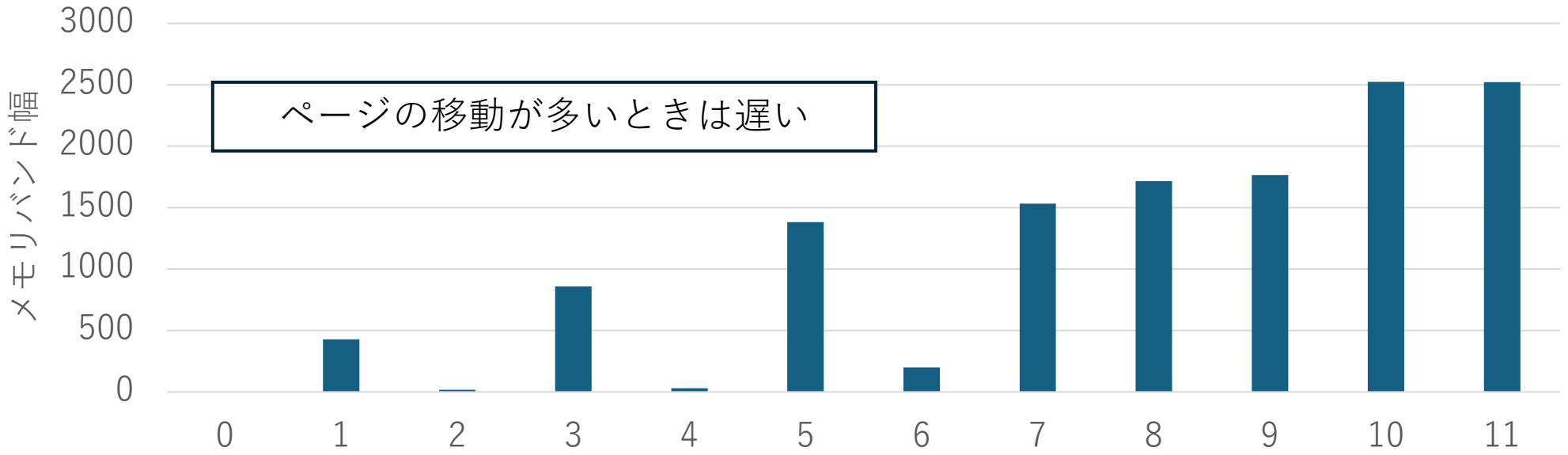
CPUで初期化する場合
この順に遷移

GPUに移動したあとCPUから幾らアクセスしてもCPUには戻らない。
※move_pages()関数でCPUに戻せる。

ページの移動状況と計測されるメモリバンド幅

256MB=4096ページ





ReadとWriteのバランス

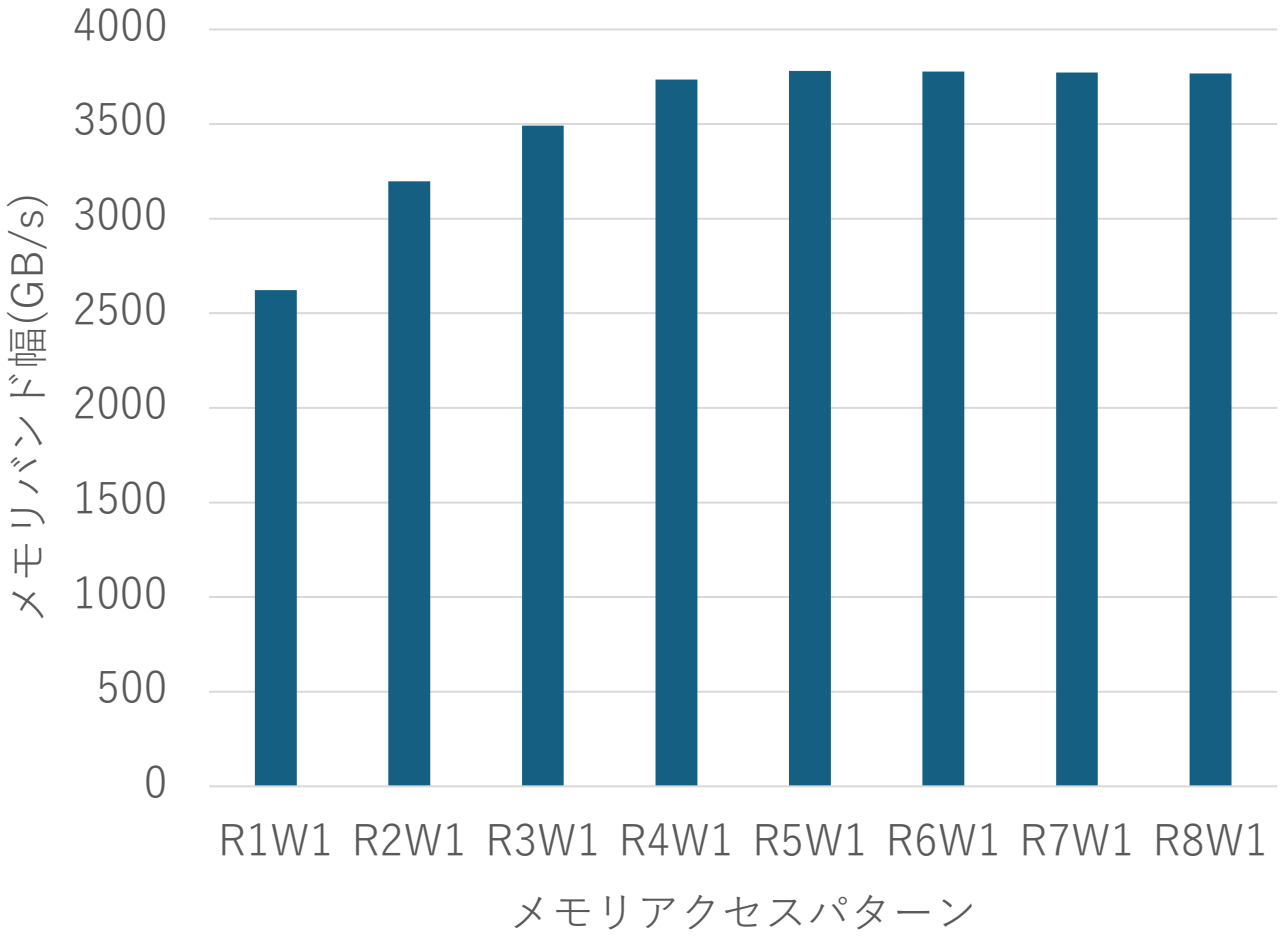
一般的にメモリの書き込みは重い

a0[i] = a0[i] + a1[i] + a2[i] …

Write　　Readアクセス

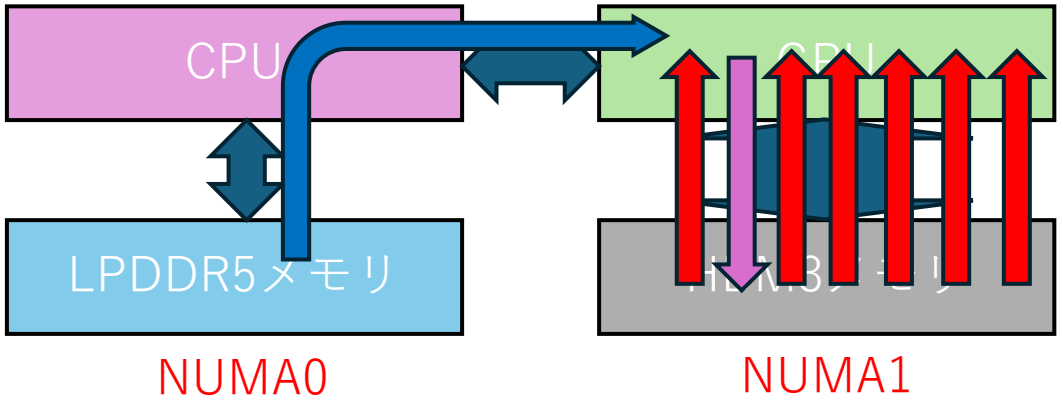
ReadとWriteのバランスを変えると
右図のように４：１で上限に到達

理論バンド幅4TB/sにかなり近い



ホストメモリの併用～よりバンド幅を求める方に～

GPUから両方のメモリにアクセスする



NVIDIA GH200の構成

a0[i] = a0[i] + a1[i] + a2[i] …

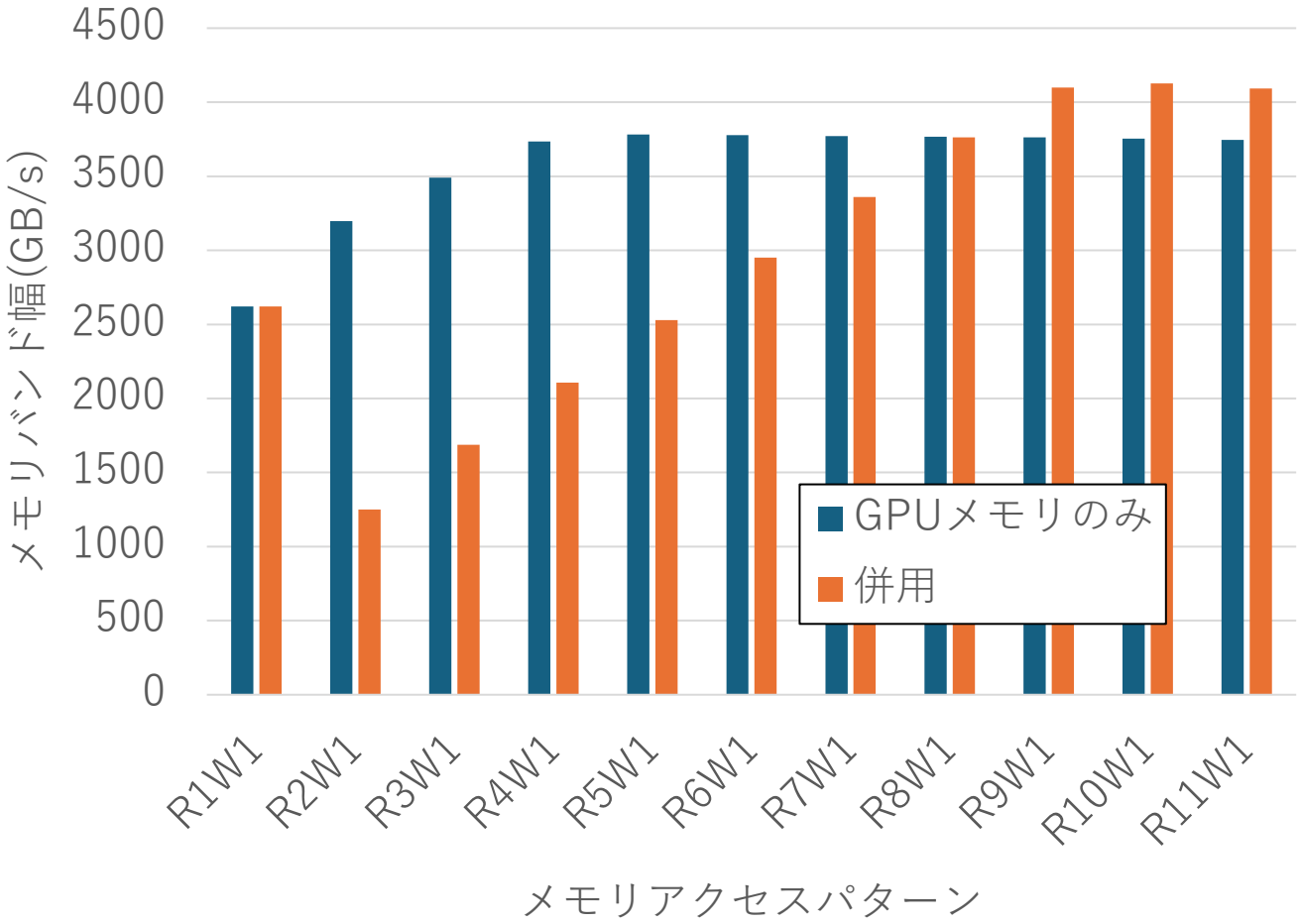
このa1[i]をホストメモリに置く

cudaHostAlloc()でpinned memoryを確保

レイテンシは大きい

GPUメモリ: 145ns、CPUメモリ: 620ns

同様にCPUで700GB/sくらい出すことも可。



理論値から算出すると9:1がベスト。実測では10:1。

GH200のUnified Memoryの注意点

- マイグレーションの完了には時間がかかる
  - 性能測定には注意
- GPUに移動するとCPUに戻らない
  - move_pages()などで戻すことは可。GPUメモリが溢れても戻る。
  - Managed MemoryでGPUメモリに固定することも可

- Fortranの場合等はコンパイラオプションで一括指定(HPC SDK全般)
  - CPUからもGPUからもアクセスするならUnified Memory
  - GPUからのみになるならManaged Memoryでもよい
  - CPUからのレイテンシが問題になるデータはGPUから触らない

MI300Aのメモリバンド幅

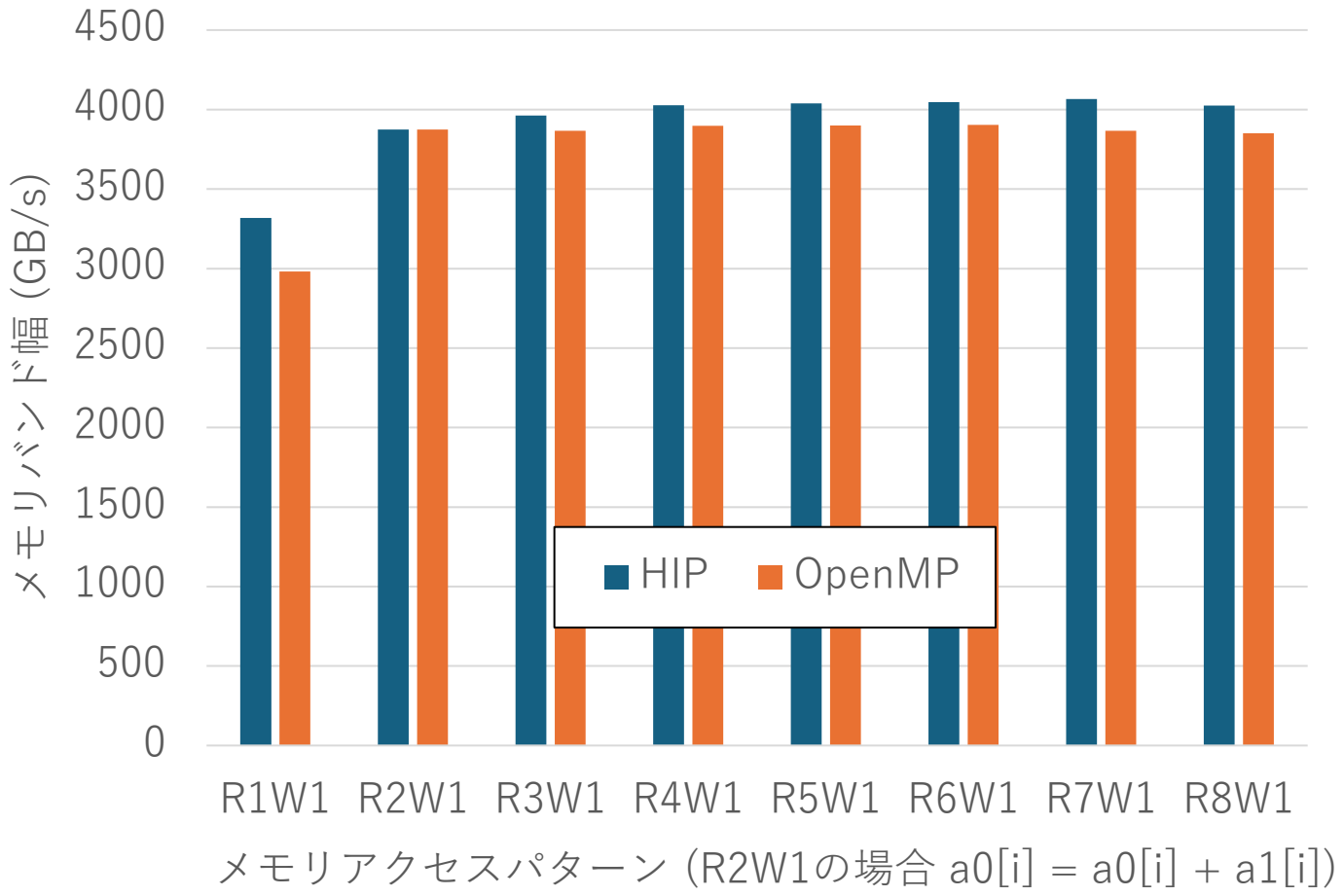
最初のGPUアクセスは遅い
（ページテーブル作成のため）

Writeが少ない方が性能が出る
理論ピーク5.3TB/sの75%

OpenMP版はHIPに若干劣る
グリッドは(N/256)x256で共通

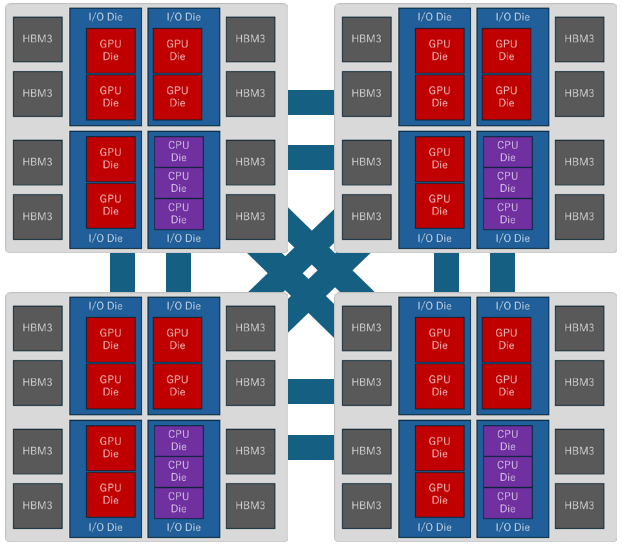
Transparent Huge Pageで自動的にHuge Pageが割り当て
hipMallocManaged()やhipMalloc()も可
hipHostMalloc()だと遅い

CPUでは260GB/s程度

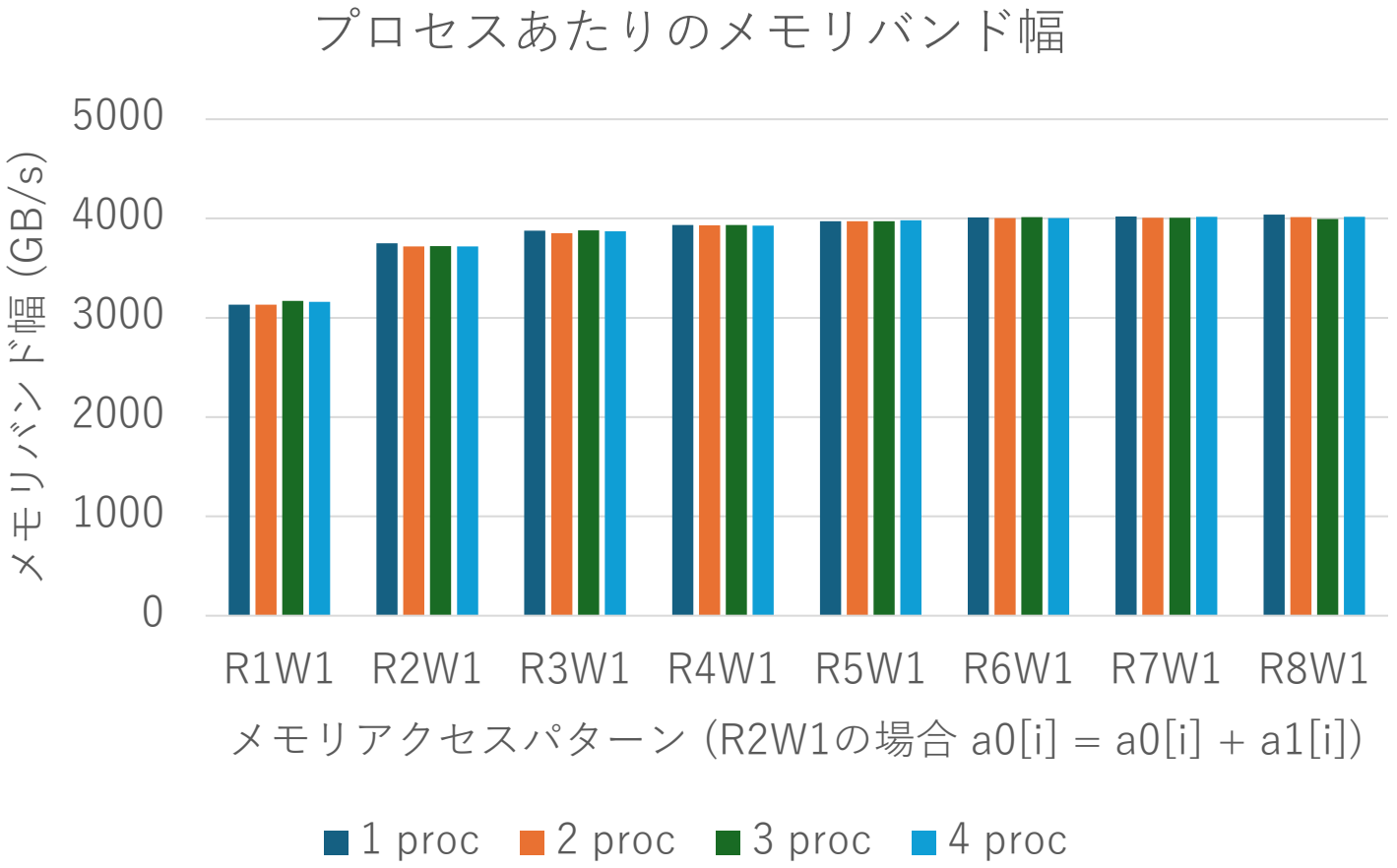


NUMAノードの同時使用時のメモリ性能

MI300Aのサーバ製品は4APU搭載
CPU用メモリが4NUMAノード構成



各APIに1MPIプロセスを実行



同じノード内の各APUで同時にそれぞれのメモリにアクセスしても性能劣化はほぼない。

OpenACCによるGPU化の普及

対象プログラムがループ構造なら容易

GPUで速くならないことはよくある

- 並列性不足
- 単純並列ではない
- ホスト・デバイス間転送が多発
- メモリレイアウトが適切でない
- １反復で使用するデータ（変数）が多すぎる

  CPUはL1キャッシュを活用

  GPUでは多数のスレッドがレジスタ退避を行うと大ダメージ

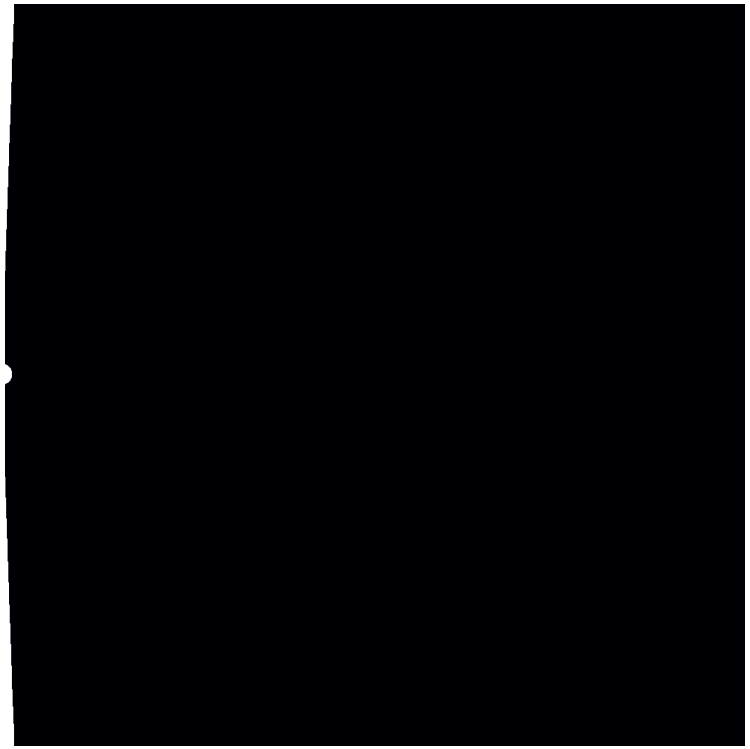
  　レジスタ退避を最小化するコンパイラの最適化の質に依存

ブラックホール降着円盤コードINAZUMA

- ３次元極座標空間　⇒３重ループの並列化は容易
- Fortranコード
- 物理計算が重く、多数の変数を使用する関数

  real(8),dimension(1:4,1:4)のような配列変数

  OpenACCのprivate節で指定(スレッド毎に異なる値)



コンパイル時に-Minfo=accelを指定すると以下の表示

68, Local memory used for z_a_23,z_a_21,z_a_89,…

なんと127個の変数が並ぶ

メモリアクセスしながら計算するので実際に遅い！

配列変数の分解

Local memoryを割り当てる変数に配列変数が多い
⇒配列でなくせばいい？
beta(1:3)はbeta_1,beta_2,beta_3に分解（変数宣言＆OpenACCのprivate節）
Fortranの範囲処理は展開
MATMUL関数、DOT_PRODUCT関数も展開
※範囲処理の展開以外はマクロにより可読性を保つ

結果としてLocal memoryに割り当てられなくなる
ただ、レジスタが溢れるような長いカーネルにこの変換は厳しい

変換の自動化（試作経験から）

方針

（何故か）単一ファイルを変換

幾つかの仮定が必要

- OpenACCのprivate変数はサブルーチン内で定義
  モジュール変数だと配列かどうか、サイズなどが取得不可
- 追加の一時変数の型は固定（全部REAL(8)等）
  モジュール変数の型が不明なので
- パラメータ変数の情報を与える
  配列サイズやインデックスに使われている整数の値等

最適化の効果の評価

どの変換が効果があるのかを検証するため３段階を用意

	Original	Extract only	Full scalar
関数呼び出しのインライン展開	○	○	○
短いループの展開	○	○	○
プライベート配列変数への動的参照の消去	○	○	○
配列処理の展開		○	○
MATMUL,DOT_PRODUCTの展開		○	○
プライベート配列変数のスカラ化			○

Extract onlyはFull scalarの一部変換を無効化

範囲処理や組み込み関数の展開は行うが、変数は配列のまま

カーネルの実行時間

nsys nvprofで取得（単位は秒）

Originalで実行時間が長い順に4個

	Original	Extract only	Full scalar	
hlle_gr_96_gpu	11.832	0.027	0.016	**x710**
cal_prad_ksd_omp_308_gpu	3.045	0.002	0.002	**x1200**
cal_eddington_tensor_omp_564_gpu	1.912	0.016	0.016	**x115**
utoprim_sc_pmhd_it_68_gpu	0.109	0.072	0.068	**x1.6**

Extract onlyだけで大幅な短縮

Full scalar化でさらに向上するものもある

コンパイル時のカーネル情報(-gpu=ptxinfo)

	Original	Extract only	Full scalar
hlle_gr_96_gpu	Stack: 928, Spill:396+396 Reg:255	Stack:408,Spill:528+520 Reg:255	Stack:560,Spill:964+800 Reg:255
cal_prad_ksd_omp_308_gpu	Reg:214	Reg:214	Reg:214
cal_eddington_tensor_omp_564_gpu	Reg:167	Reg:168	Reg:168
utoprim_sc_pmhd_it_68_gpu	Stack:552,Spill:724+2768 Reg:255	Stack:680,Spill:972+2868 Reg:255	Stack:560,Spill:752+1936 Reg:255

Spill:X+Y => X spill store & Y spill load

使用レジスタ数は減らない。StackやSpillはむしろ増える。
Full scalar化でさらに向上していたのはRegをフルに使っているカーネルのみ。

カーネルのメモリアクセス

ncu –metrics dram__bytes_read,dram__bytes_writeで取得

	Original	Extract only	Full scalar
hlle_gr_96_gpu	READ: 1.81GB WRITE: 3.00GB	READ: 1.24GB WRITE: 1.72GB	READ: 0.92GB WRITE: 0.60GB
cal_prad_ksd_omp_308_gpu	READ: 513MB WRITE: 278MB	READ: 441MB WRITE: 180MB	READ: 441MB WRITE: 180MB
cal_eddington_tensor_omp_564_gpu	READ: 3.41GB WRITE: 481MB	READ: 3.15GB WRITE: 175MB	READ: 3.14GB WRITE: 175MB
utoprim_sc_pmhd_it_68_gpu	READ: 1.60GB WRITE: 1.67GB	READ: 1.98GB WRITE: 1.98GB	READ: 1.57GB WRITE: 0.69GB

実際にメモリアクセス量は減っている（一部例外あり）

Local memoryへのアクセス命令数も減っている

コンパイラのメッセージやプロファイラ出力はある程度参考になる

MI300Aの性能評価：ボルツマン方程式

ボルツマン方程式における衝突項を求めるベンチマークコードに適用

$$\frac{\partial f}{\partial t} + \boldsymbol{v} \cdot \frac{\partial f}{\partial \boldsymbol{x}} + \frac{\mathrm{d}\boldsymbol{v}}{\mathrm{d}t} \cdot \frac{\partial f}{\partial \boldsymbol{v}} = Q(f)$$

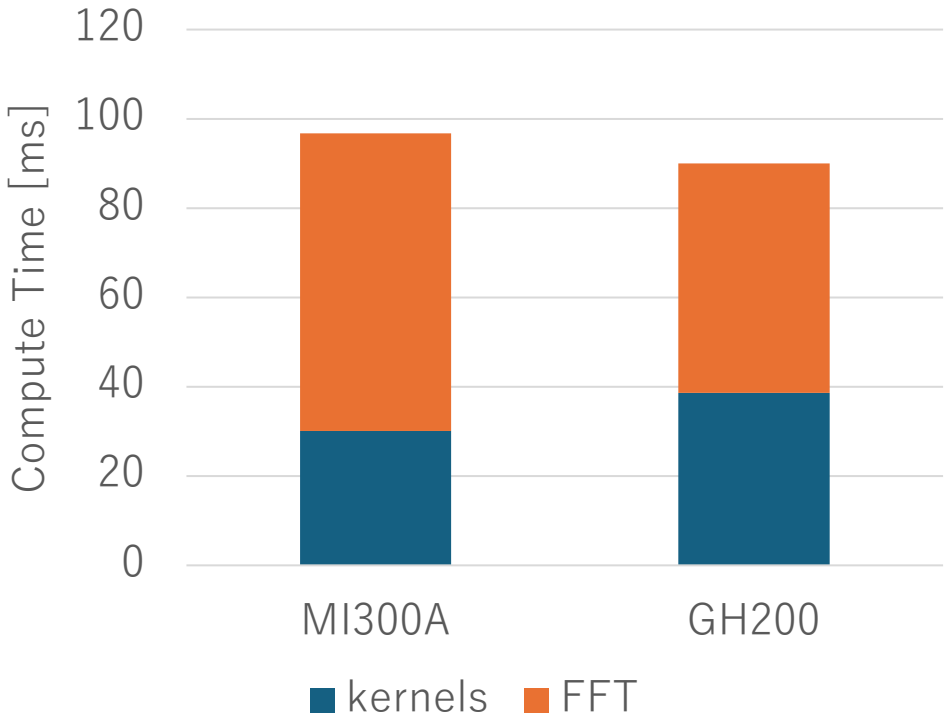
$$Q(f)(\boldsymbol{v}) = \int_{\mathbb{R}^3} \int_{S^2} \left[f(\boldsymbol{v}') f(\boldsymbol{v}_1') - f(\boldsymbol{v}) f(\boldsymbol{v}_1) \right] |\boldsymbol{v} - \boldsymbol{v}_1| \frac{\mathrm{d}\sigma}{\mathrm{d}\Omega} \, \mathrm{d}\Omega \, \mathrm{d}\boldsymbol{v}_1$$

計算の主たる部分はFFTによる畳み込み

小さい3次元FFTを大量にBatchedで計算

CUDA CからhipifyでHIPに自動変換

FFTライブラリの呼び出しも変換される



kernel部分は速いがFFT部分で遅い

32^3のサイズはNVIDIAに有利

MI300Aの性能評価：ブラックホール降着円盤

INAZUMAコード（private配列変数のスカラ化済み）

[方針] OpenACC関連をOpenMP target化

OpenACC	OpenMP (AMD向け)
!$acc parallel loop collapse(3) vector_length(32)	!$omp target teams distribute parallel do collapse(3) thread_limit(64)
!$acc enter data create(u) copyin(p)	!$omp target enter data map(alloc:u) map(to:p)

!$acc kernels
  temp_irfac(jms:jme) = irfac(jms:jme)
!$acc end kernels
	!$omp target teams distribute parallel do
	  do j=jms, jme
	  temp_irfac(j) = irfac(j)
	  end do

!$omp target workshareは使えない

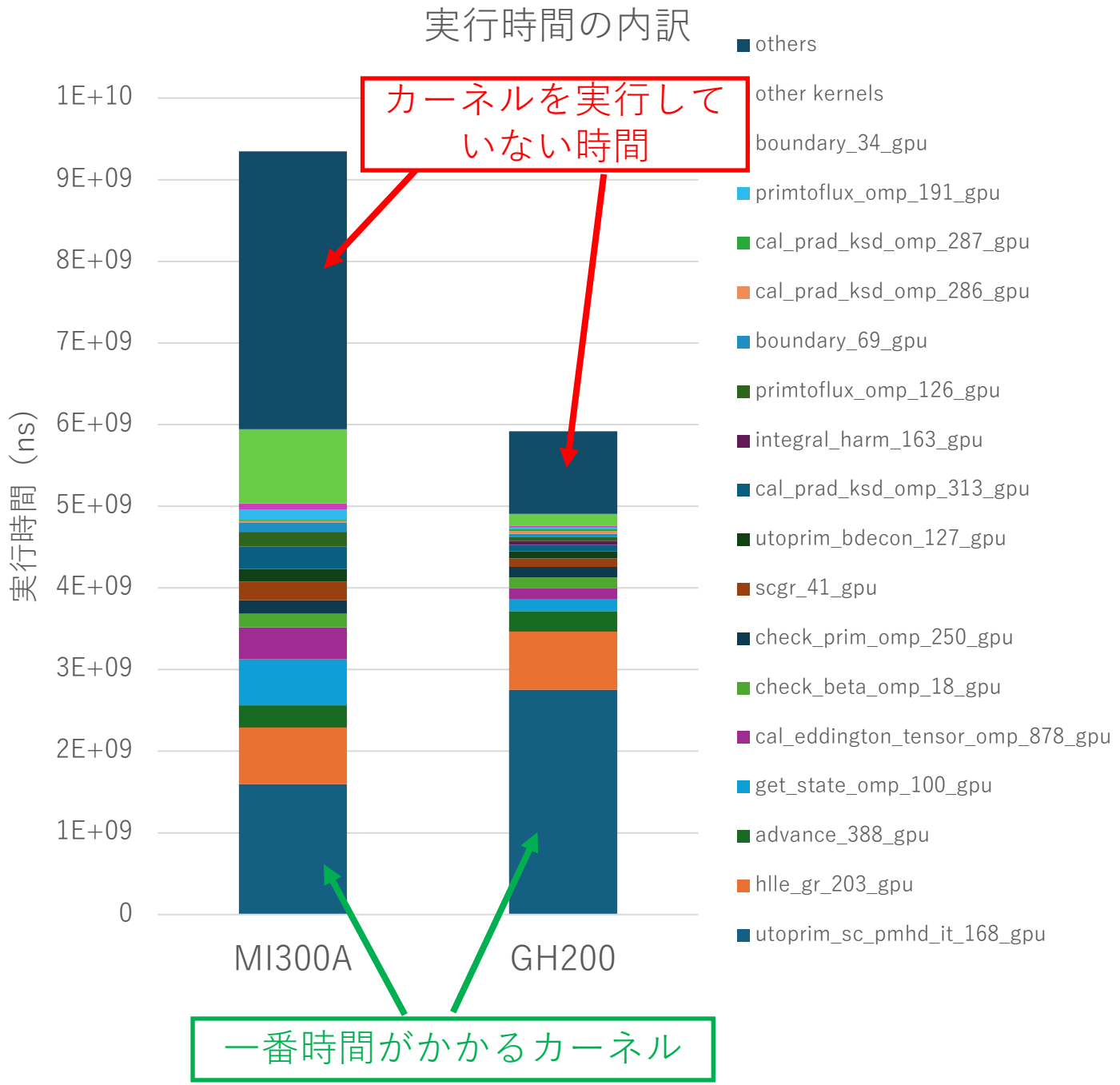
実行結果

- GH200で時間がかかっていた処理は同等か速く
- 小さいカーネルは一部遅い
- カーネル以外に長い時間

カーネル呼び出し時間

GH200: 7.23 msec

MI300A: 11.34 msec



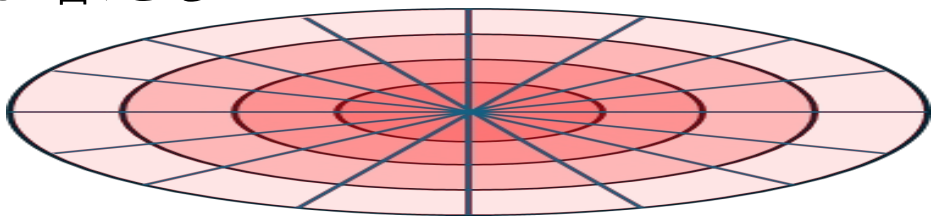
LATを無効にした場合

Local Adaptive Time-stepping (LAT)

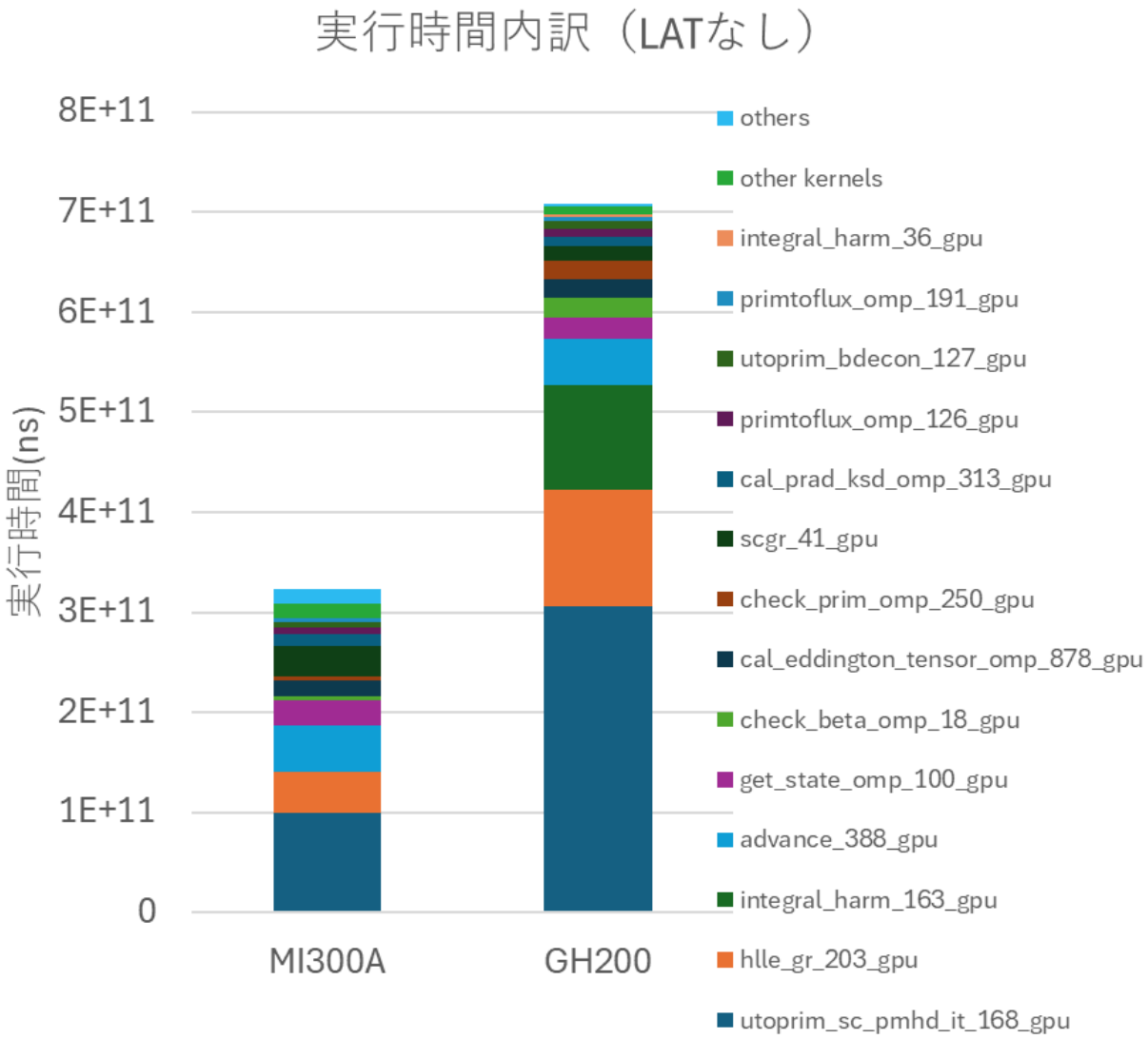
ブラックホールの中心から

- 近くはタイムステップを細かく
- 遠くはタイムステップを荒く

演算量は減るが、カーネル呼び出し回数は増える



LATを無効にすると演算量が増えるが並列度も増え、カーネル呼び出しが減。



HAIRDESC GPUプログラミング教材（第1版）

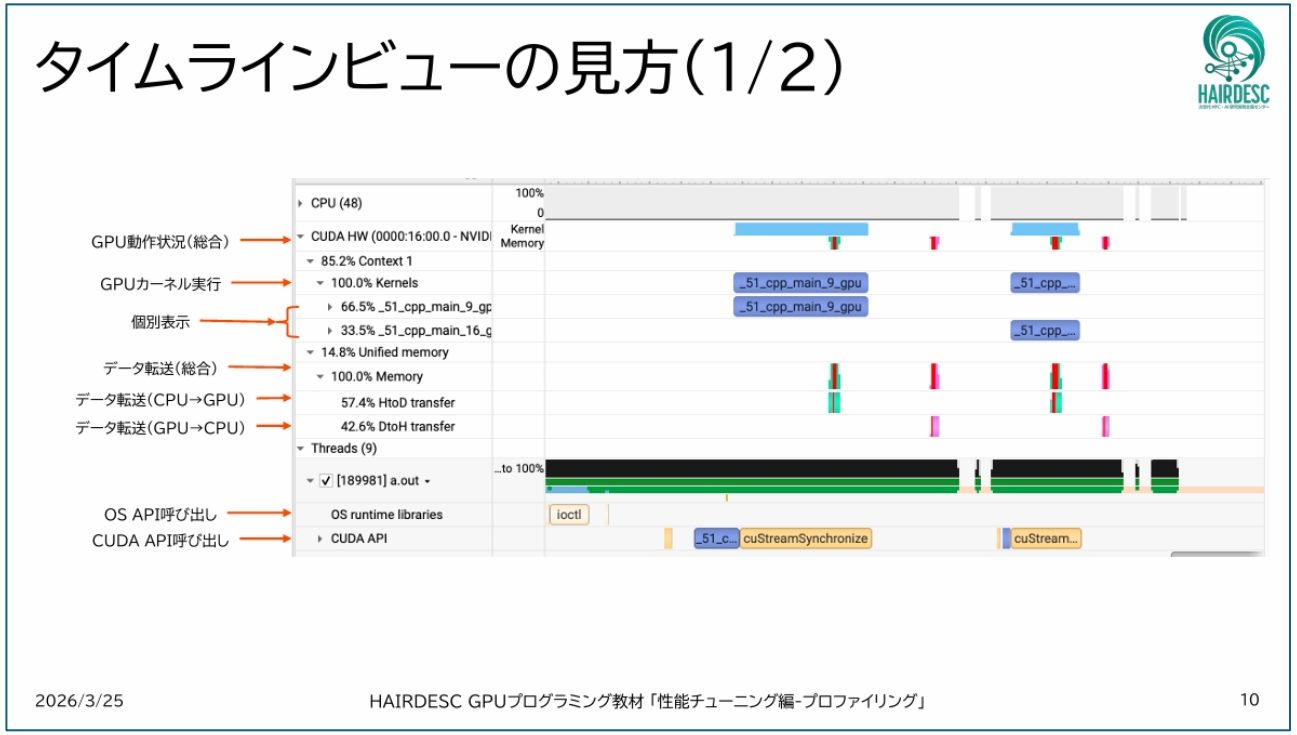
- 代表機関・中核機関のWGメンバーで議論
- 筑波大はパフォーマンスチューニング編の「プロファイル」の原案を担当
  - 妥当な性能か検証
  - 性能が出ないときに問題点を探る

    どのカーネル？

    並列度は足りる？

    メモリネック？

    余計な転送はない？



まとめ

- Unified Memoryの性能特性や挙動の解析
- OpenACCプログラムのレジスタ溢れ対策の１案
- AMD MI300Aの評価（GH200との比較）
- GPUプログラミング教材