Massive parallel programming on Graphics Processing Units and Applications (part 3)

Lokman Abbas-Turki
lokmane.abbas_turki@sorbonne-universite.fr

This project has received funding from the European High-Performance Computing Joint Undertaking under grant agreement No 101051997

October 2023

EUMaster4HPC

Plan

Cache memory on GPU

- Real cache: shared, L2
- Virtual cache: constant, texture, local memory

Communication and reduction

- Using shared memory
- Using registers

Further optimizations

- Pinned memory and zero copy
- Concurrency and asynchronous data transfer

Real applications

- From Monte Carlo to nested Monte Carlo
- Partial Differential Equation simulation

EUMaster4HPC

Plan

Cache memory on GPU

Real cache: shared, L2
Virtual cache: constant, texture, local memory

Communication and reduction

Using shared memory
Using registers

Further optimizations

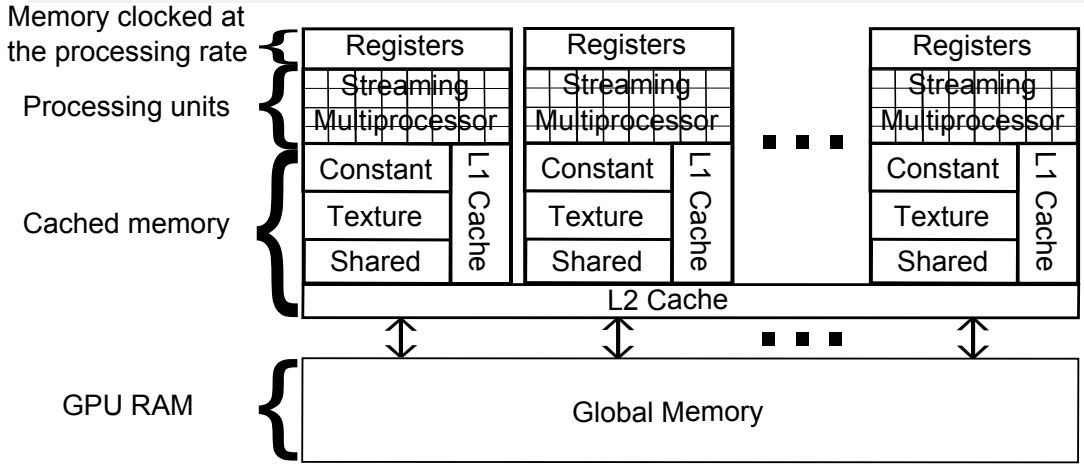
Pinned memory and zero copy
Concurrency and asynchronous data transfer

Real applications

From Monte Carlo to nested Monte Carlo
Partial Differential Equation simulation

EUMaster4HPC

Shared: Second fastest memory



Shared memory

- Cached memory visible to all threads of the same block
- Has a lifetime of a kernel
- Static allocation of arrays: `__shared__ float A[100];`
- Dynamic allocation of arrays: `extern __shared__ float A[];`
  kernel call: `myKernel<<<..., ..., 100*sizeof(float)>>>(...);`

High bandwidth

- Divided into equally sized memory modules (banks)
- Any memory load/store of n addresses ($n \leq 32$) in n distinct memory banks can be performed simultaneously
- Multiple accesses to the same memory bank are serialized, except for the same memory location accessed by warp threads (broadcast)

Increase the size of shared in GPUs

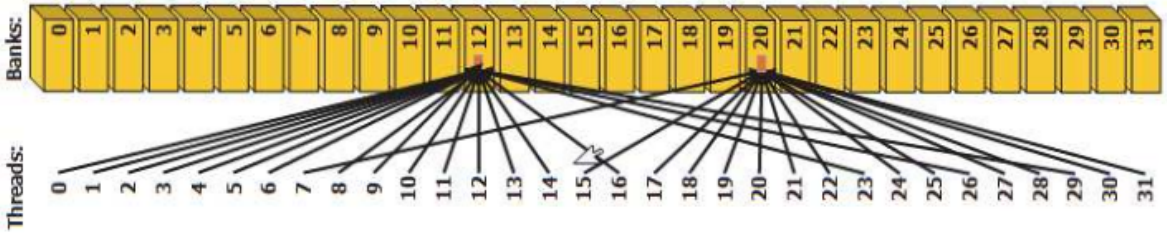
with `cudaFuncAttributeMaxDynamicSharedMemorySize` in `cudaFuncSetAttribute()`

EUMaster4HPC

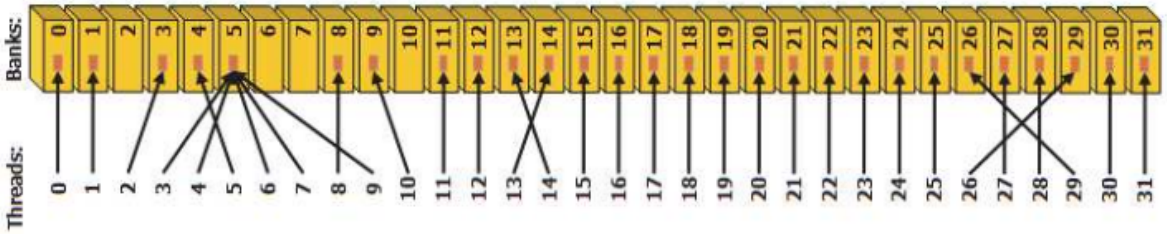
Sarr is an array in the shared memory

Figure from CUDA programming guide, Nvidia

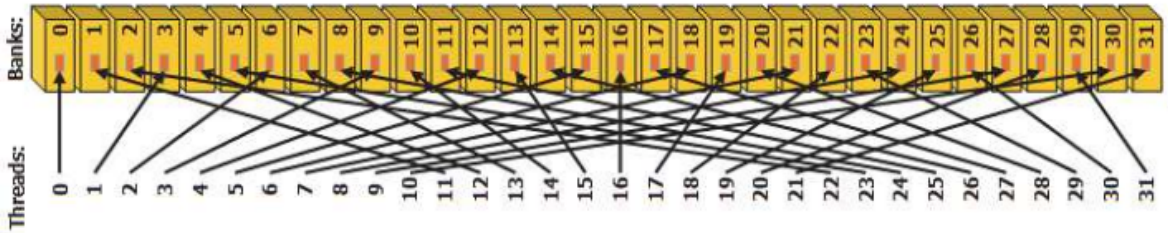
no bank conflict (broadcast)



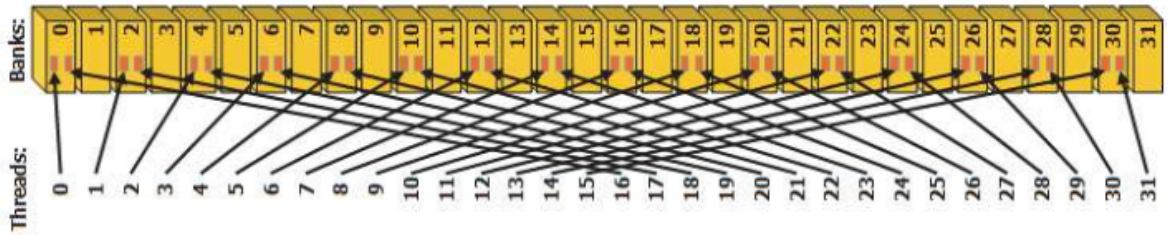
no bank conflict



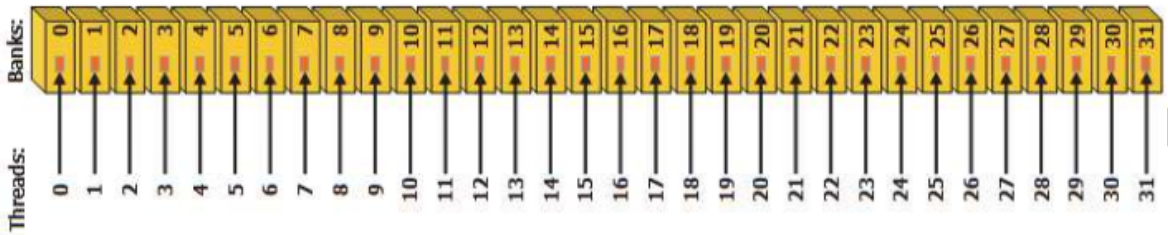
Sarr[3*threadIdx.x] no bank conflict



Sarr[2*threadIdx.x] two-way bank conflict

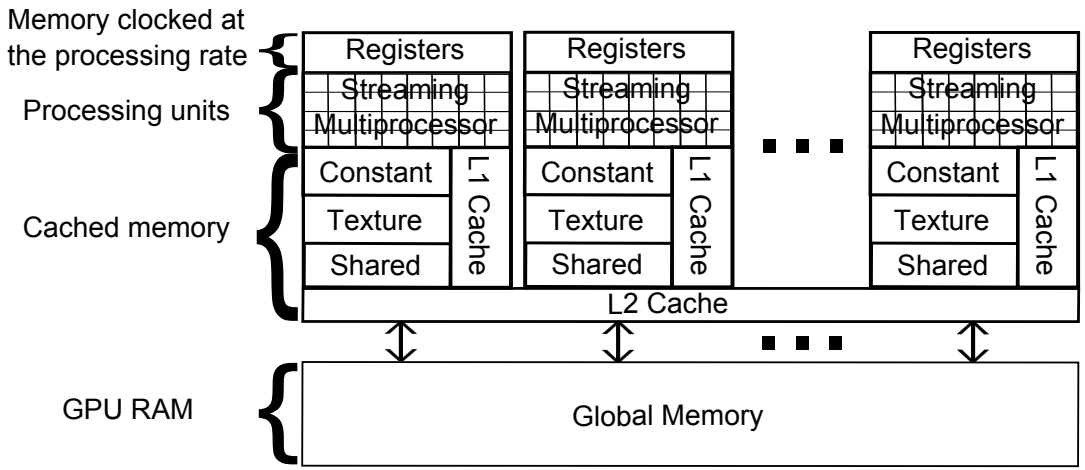


Sarr[threadIdx.x] no bank conflict



EUMaster4HPC

L2 cache: for global memory access



3 types of caching Starting with CUDA 11.0 and compute capability 8.0, we can specify

`cudaAccessPropertyNormal = 0` Normal cache persistence, removes persisting status of prior access

`cudaAccessPropertyStreaming = 1` Less likely to persit in cache, accesses are preferentially evicted

`cudaAccessPropertyPersisting = 2` More likely to persist in cache, preferentially retained in cache

Caching size Starting with CUDA 11.0 and compute capability 8.0, we can specify

`l2CacheSize` The size of L2 cache on the GPU.

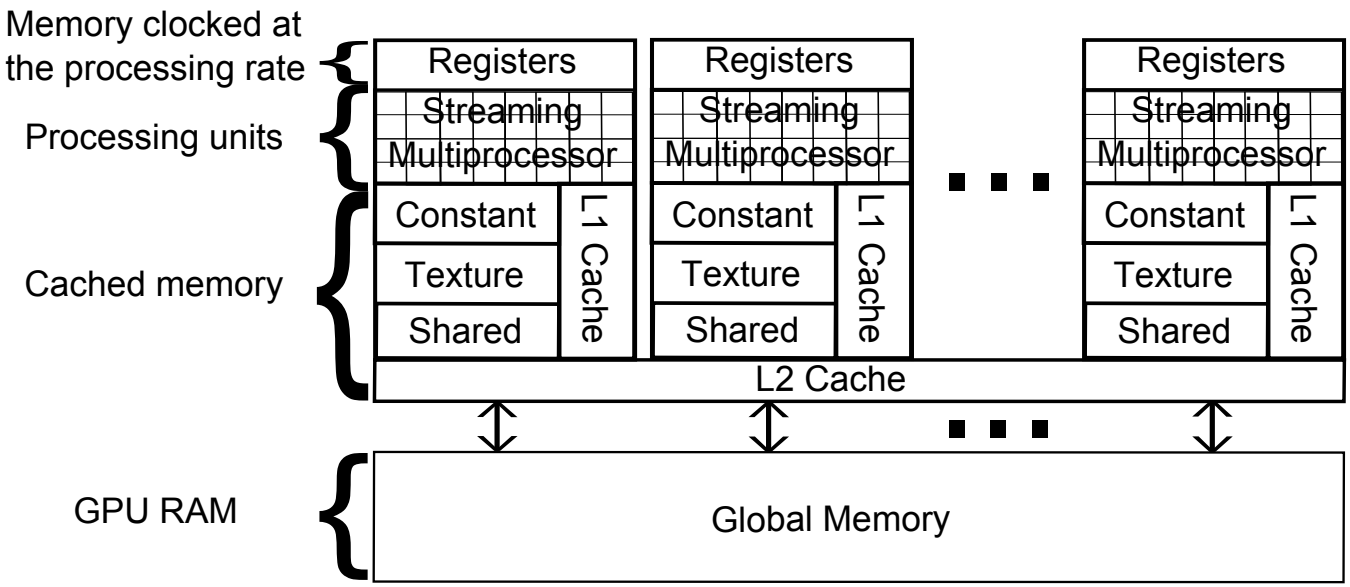
`persistingL2CacheMaxSize` The maximum size that can be set-aside for persisting memory accesses

`accessPolicyMaxWindowSize` The maximum size of the access policy window

Tuning `hitRatio` If `hitRatio=1`, the window size should fit the L2 cache set-aside.
For window sizes > `persistingL2CacheMaxSize`,
one should tune `hitRatio<1`

EUMaster4HPC

Read from device memory only: constant memory

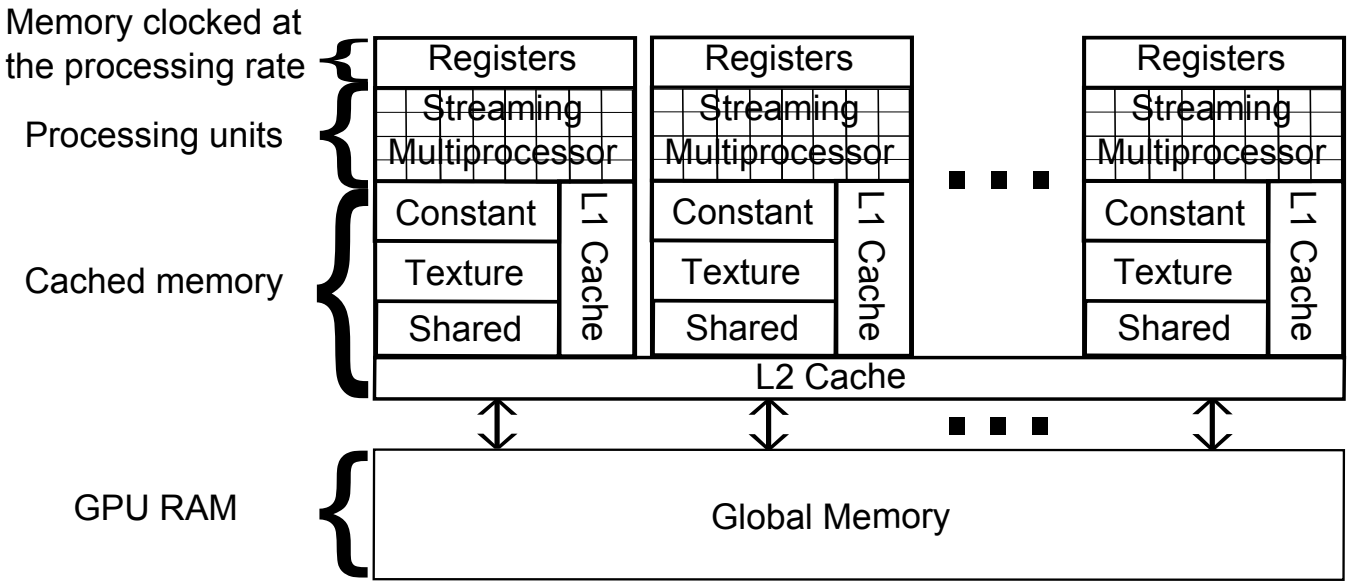


Constant memory

- Cannot be changed by threads
- Resides in device memory, cached in the constant cache
- Accesses to different addresses by threads within a warp are serialized
- Has the lifetime of the CUDA context in which it is created (global declaration)
- Values can be sent from the host using `cudaMemcpyToSymbol`
- Values can be read from the host using `cudaMemcpyFromSymbol`

EUMaster4HPC

Read from device memory only: texture memory



Texture memory

- Cannot be changed by threads
- Resides in device memory, cached in the texture cache
- Optimized for 2D spatial locality: best performance for threads of the same warp that read from addresses that are close together
- Read-only for the entire lifetime of the kernel

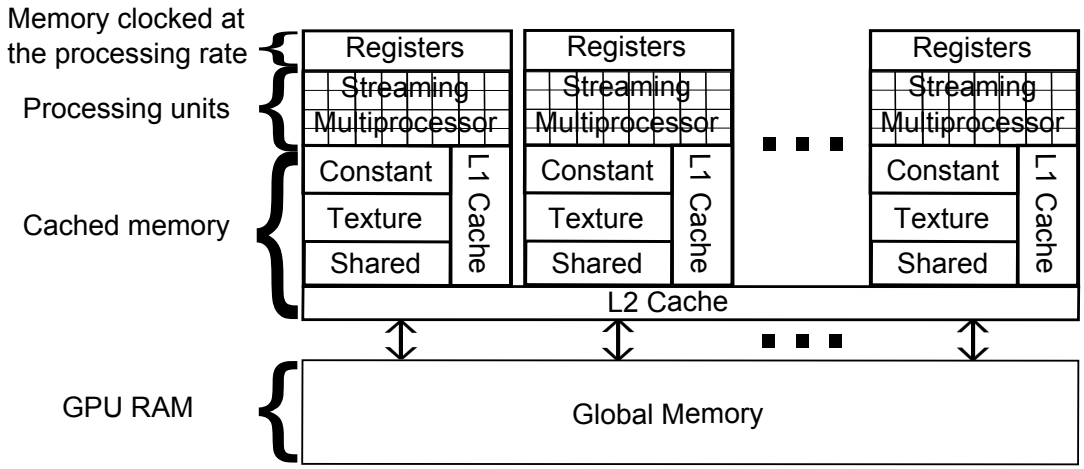
Better alternatives

Verbose code, it is usually better to use

- `__ldg` for read-only array with a larger size than constant memory
- shared memory or registers (seen later) for spatial locality

EUMaster4HPC

Local memory: if not possible to use registers or shared



Local memory

- Local with respect to each thread
- Has the lifetime of the kernel
- Resides in device memory, cached if possible in L1/L2
- Otherwise has the same high latency and low bandwidth as global memory
- nvcc compiler use it when there is insufficient register space for variables

Example In a kernel or a device function

- `float a[10]; //means that each thread has its own array`
- `a[0] = 0; //means that each thread sets the value of its own a[0]`
- Accesses are fully coalesced as long as all threads in a warp access the same relative address: the same index in an array variable or the same member in a structure variable

EUMaster4HPC

Plan

Cache memory on GPU

Real cache: shared, L2
Virtual cache: constant, texture, local memory

Communication and reduction

Using shared memory
Using registers

Further optimizations

Pinned memory and zero copy
Concurrency and asynchronous data transfer

Real applications

From Monte Carlo to nested Monte Carlo
Partial Differential Equation simulation

EUMaster4HPC

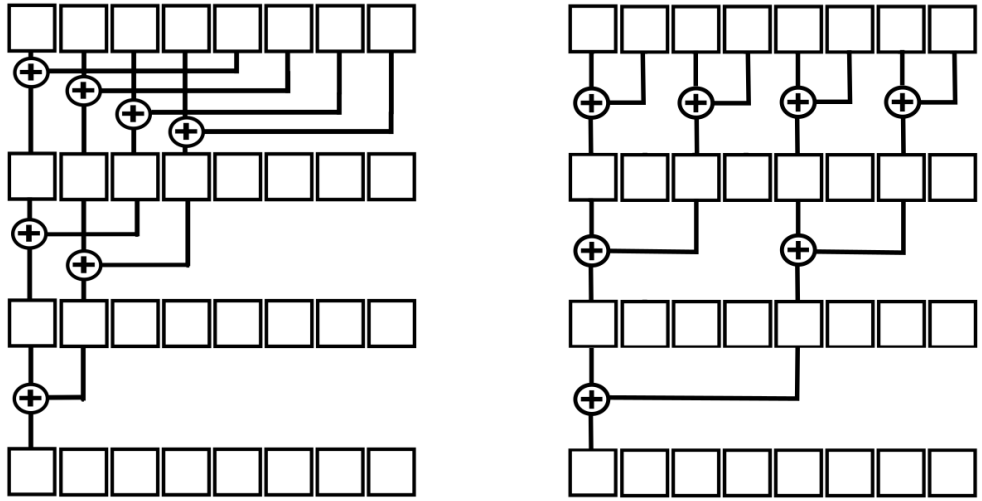
Avoiding bank conflicts

Some facts

- Threads can access to any memory space of the shared memory of their block
- The synchronization barrier `__syncthreads();` ensures that threads of the same block wait for all other threads of the same block.
- Threads of different blocks cannot exchange values within the same kernel

Dot product

- Store the product result in the shared memory then perform a reduction using the following scheme (left) with a `__syncthreads();` at each step



- `atomicAdd` is used for the reduction through blocks
- What happens when `atomicAdd` is used for the whole sum?

EUMaster4HPC

```
1  __global__ void ShaAtom_k(float* A, float* B, float* C) {
2
3      int idx = threadIdx.x + blockIdx.x * blockDim.x;
4      int i;
5      __shared__ float sC[NTPB];
6
7      sC[threadIdx.x] = A[idx] * B[idx];
8      __syncthreads();
9
10     i = blockDim.x / 2;
11     while (i != 0) {
12         if (threadIdx.x < i) {
13             sC[threadIdx.x] += sC[threadIdx.x + i];
14         }
15         __syncthreads();
16         i /= 2;
17     }
18
19     if (threadIdx.x == 0) {
20         atomicAdd(C, sC[0]);
21     }
22 }
```

EUMaster4HPC

```
1  ////////////////Bad alternative//////////////////////
2  int i = blockDim.x / 2;
3  while (i != 0) {
4      if (threadIdx.x < i) {
5          sC[threadIdx.x] += sC[threadIdx.x + i];
6          __syncthreads();
7      }
8      i /= 2;
9  }
10
11 ////////////another bad example//////////////////////
12 if (/*boolean that depends on each thread*/) {
13     /*some operations*/
14     __syncthreads();
15 }
16
17 ////////////another bad example//////////////////////
18 for (int i = 0; i < threadIdx.x * threadIdx.x; i++) {
19     /*some operations*/
20     __syncthreads();
21 }
```

EUMaster4HPC

Some facts

- Threads in the same warp, called lanes, can access to any register associated to their warp
- Lanes are synchronized with `__syncwarp(unsigned mask=0xffffffff)` more local barrier, where `mask` specifies the lanes involved
- Functions prefixed with `__shfl` and suffixed with `__sync` are needed for this communication. These functions ensure lanes converged before the intrinsic operation is executed (using `__syncwarp(mask)` not required)

Syntax example

```
float __shfl_down_sync(unsigned mask, float var,
unsigned int delta, int width)
```

- `var` is the value to be communicated
- `delta` is a lane translation index
- `width` is the number of involved threads $\leq$`warpSize=32`

Testing this with 1 block and 32 threads

```
1 __global__ void print_k() {
2
3     int lane = threadIdx.x % 32;
4     int i = __shfl_down_sync(0xffffffff, lane, 2, 32);
5     if (lane < 2 || lane>28 || (lane < 18 && lane>12))
6         printf("(%d, %d); ", lane, i);
7 }
```

it provides (0, 2); (1, 3); (13, 15); (14, 16); (15, 17); (16, 18); (17, 19); (29, 31); (30, 30); (31, 31);

if `width=16`

it provides (0, 2); (1, 3); (13, 15); (14, 14); (15, 15); (16, 18); (17, 19); (29, 31); (30, 30); (31, 31);

EUMaster4HPC

```
1  __global__ void RegAtom_k(float* A, float* B, float* C) {
2
3      int idx = threadIdx.x + blockIdx.x * blockDim.x;
4      int lane = threadIdx.x & 0x1f;
5      int i;
6      float loc;
7      __shared__ float sC[32];
8
9      loc = A[idx] * B[idx];
10
11     i = 16;
12     while (i != 0) {
13         loc += __shfl_down_sync(0xffffffff, loc, i, 32);
14         i /= 2;
15     }
16
17     if (lane == 0)sC[threadIdx.x / 32] = loc;
18     __syncthreads();
```

EUMaster4HPC

```
20      i = blockDim.x / (2 * 32);
21      while (i != 0) {
22          if (threadIdx.x < i) {
23              sC[threadIdx.x] += sC[threadIdx.x + i];
24          }
25          __syncthreads();
26          i /= 2;
27      }
28      if (threadIdx.x == 0) {
29          atomicAdd(C, sC[0]);
30      }
31  }
```

Benefits

- Makes the shared memory available for other tasks
- Registers are faster
- Code without explicit synchronization between threads (performed by Shuffle functions)

EUMaster4HPC

Plan

Cache memory on GPU

Real cache: shared, L2
Virtual cache: constant, texture, local memory

Communication and reduction

Using shared memory
Using registers

Further optimizations

Pinned memory and zero copy
Concurrency and asynchronous data transfer

Real applications

From Monte Carlo to nested Monte Carlo
Partial Differential Equation simulation

EUMaster4HPC

On the host: pageable memory vs. locked memory vs. mapped memory

Standard host allocation

- Memory space essentially limited by the machine's RAM
- The OS controls the memory space with the possibility to put data on disk
- The data transfer is the slowest it can get

Locked or pinned allocation

- Memory space much smaller then the machine's RAM
- The OS loses some control on the memory space
- The data transfer is faster then standard allocation

Mapped or zero-copy allocation

- Memory space much smaller then the machine's RAM
- The OS loses some control on the memory space
- Double pointers: One used by CPU and the other one used by GPU.
- The **implicit** data transfer is the fastest it can get

Locked vs. mapped

- Mapping increases virtually the size of the GPU RAM
- The data usually used by the GPU should be stored in its RAM and not on mapped memory

EUMaster4HPC

How to implement?

Locked memory

- Make sure that the memory space to be locked is not too big
- Replace `malloc` by `cudaHostAlloc` and `free` by `cudaFreeHost`

Mapped allocation

- Make sure that the memory space to be locked is not too big
- Before calling any other function (at the beginning of main), one has to execute `cudaSetDeviceFlags(cudaDeviceMapHost)`
- The CPU allocation has to be done using `cudaHostAlloc` with `cudaHostAllocMapped` as an option.
- The GPU gets a pointer with `cudaHostGetDevicePointer`
- Make sure that the GPU had finished working on mapped memory thanks to `cudaDeviceSynchronize`.

Compared to unified memory

- When locked and mapped are in the CPU RAM, the unified is by default in the GPU RAM
- We can however modify the behavior of the unified using `cudaMemAdvise`
- Using the unified is less verbose especially for multiple GPUs or CPU+GPU architecture like GH200

EUMaster4HPC

Using different streams of type cudaStream_t

For concurrency

- Create streams with `cudaStreamCreate`
- Execute concurrent kernels with different streams. For example:

```
myKernel1<<<..., ..., ..., stream1>>>(...);
myKernel2<<<..., ..., ..., stream2>>>(...);
```

- Make sure that streams finished processing with `cudaStreamSynchronize`
- Destroy streams with `cudaStreamDestroy`

For asynchronous execution

- Use `cudaHostAlloc` for allocation on the host memory
- Create streams with `cudaStreamCreate`
- The transfer has to be done using `cudaMemcpyAsync` involving a different stream from the one used in the kernel call
- Destroy streams with `cudaStreamDestroy`

Remarks

- Asynchronous and overlapping transfers with computation is less and less an issue with the use of unified memory which can be optimized using `cudaMemPrefetchAsync`
- As GPU computing resources become very high, concurrency is increasingly necessary

EUMaster4HPC

Example from CUDA C++ Programming Guide

Instead of

```
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel <<<100, 512, 0, stream[i]>>>
          (outputDevPtr + i * size, inputDevPtr + i * size, size);
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
}
```

Rather use

```
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
    MyKernel<<<100, 512, 0, stream[i]>>>
          (outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
```

EUMaster4HPC

Plan

Cache memory on GPU
Real cache: shared, L2
Virtual cache: constant, texture, local memory

Communication and reduction
Using shared memory
Using registers

Further optimizations
Pinned memory and zero copy
Concurrency and asynchronous data transfer

Real applications
From Monte Carlo to nested Monte Carlo
Partial Differential Equation simulation

EUMaster4HPC

Pricing European $X = e^{-rT} f(S_T)$

$E(X) \approx \dfrac{X_1 + X_2 + ... + X_n}{n}$, using a family $\{X_i\}_{i \leq n}$ of i.i.d $\sim X$

- **Strong law of large numbers**:

$$P\left(\lim_{n \to +\infty} \frac{X_1 + X_2 + ... + X_n}{n} = E(X) \right) = 1$$

- **Central limit theorem**: denoting $\epsilon_n = E(X) - \dfrac{X_1 + X_2 + ... + X_n}{n}$

$$\frac{\sqrt{n}}{\sigma} \epsilon_n \to G \sim \mathcal{N}(0,1)$$

- There is a 95% chance of having $|\epsilon_n| \leq 1.96 \dfrac{\sigma}{\sqrt{n}}$

Euler scheme for Black & Scholes model

Given a time discretization sequence $t_k = kT/N$, with $k = 0, ..., N$ and $N = 100$, for $i = 1, ..., n$ we simulate

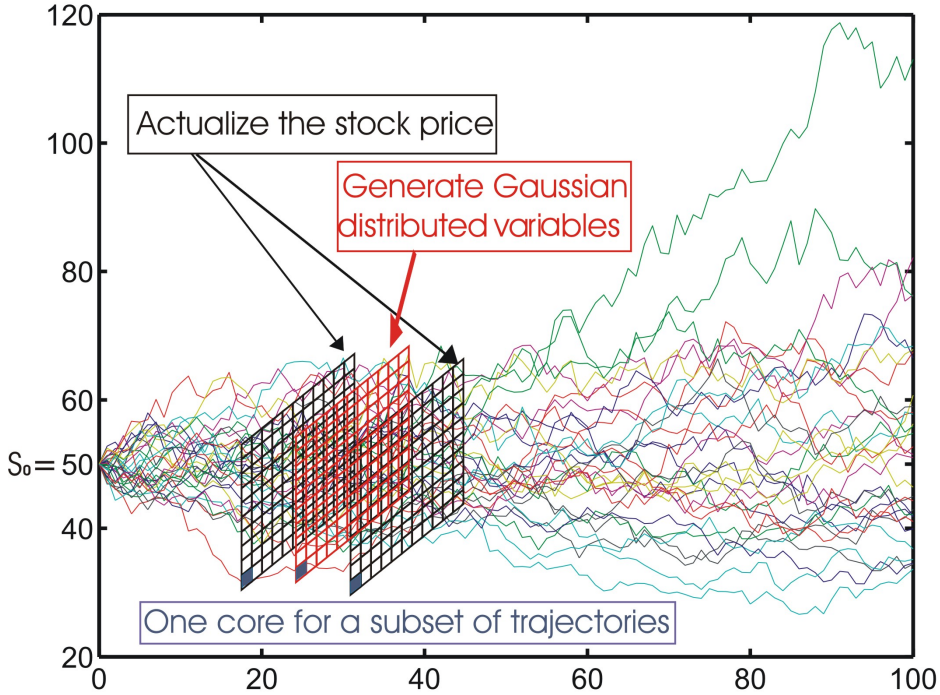
$$S^i_{t_k} = S^i_{t_{k-1}} \left[1 + rT/N + \sigma \sqrt{T/N} G^i_k \right], \quad S_0 = 50, \tag{1}$$

where the risk-free rate $r = 0.1$, the volatility $\sigma = 0.2$, and $(G^i_k)^{i=1,...,n}_{k=1,...,N}$ are independent normal random variables $\sim \mathcal{N}(0,1)$

Call Option

$f(x) = (x - K)_+ = \max(x - K, 0)$, with strike $K = S_0$ and maturity $T = 1$

EUMaster4HPC



For each time step $k < N$:

1. Random number generation (if parallelized) of G_k^i
2. Stock price actualization $S_{t_k}^i = S_{t_{k-1}}^i \left[1 + rT/N + \sigma\sqrt{T/N}G_k^i\right]$

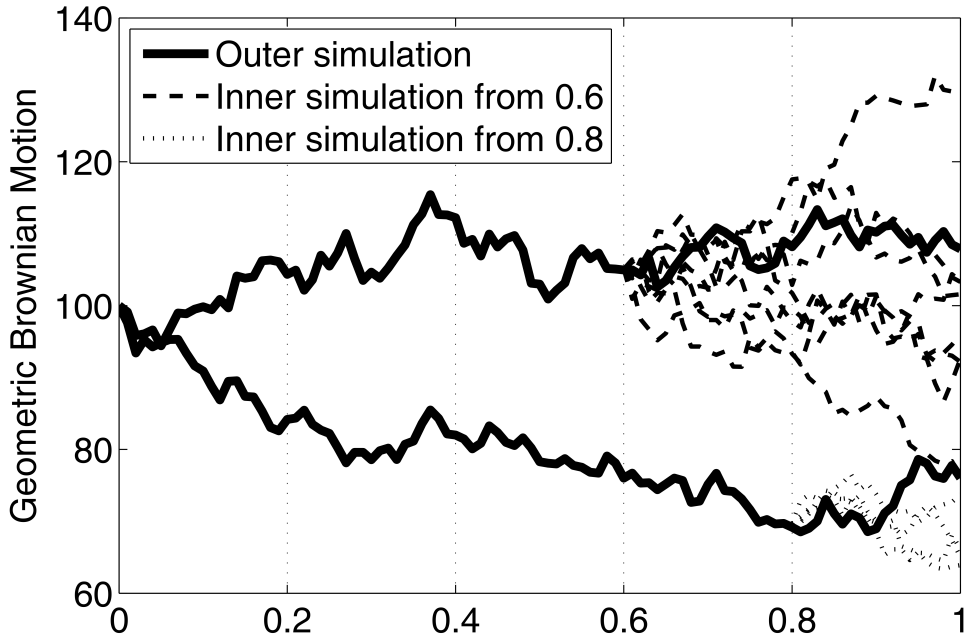
Final time step $k = N$:

3. Compute the payoff $X^i = e^{-rT}(S_T^i - K)_+$ and put it in registers
4. Perform the reduction in registers then in shared memory for the approximation $E(X) \approx \frac{X_1 + X_2 + ... + X_n}{n}$

EUMaster4HPC

Nested Monte Carlo

An example of a two stage nested Monte Carlo



Euler scheme for Black & Scholes model nested with respect to S_0

Given a time discretization sequence $t_k = kT/N$, with $k = 0, ..., N$ and $N = 100$, for $i = 1, ..., n$ and $j = 1, ..., m$ we simulate

$$S^{i,j}_{t_k} = S^{i,j}_{t_{k-1}} \left[1 + rT/N + \sigma\sqrt{T/N}G^{i,j}_k\right], \quad S^j_0 = S_{\min} + j * \Delta S, \qquad (2)$$

the risk-free rate $r = 0.1$, the volatility $\sigma = 0.2$, $(G^{i,j}_k)^{i=1,...,n;j=1,...,m}_{k=1,...,N}$ are independent normal random variables $\sim \mathcal{N}(0,1)$, $S_{\min} = 20$ and $\Delta S = 80/m$

EUMaster4HPC

Black & Scholes PDE

$$\frac{\partial F}{\partial t}(t,x) + rx\frac{\partial F}{\partial x}(t,x) + \frac{1}{2}\sigma^2 x^2 \frac{\partial^2 F}{\partial x^2}(t,x) = rF(t,x), \quad F(T,x) = f(x)$$

With $u(t,x) = e^{r(T-t)}F(t,e^x)$, we equivalently solve the PDE

$$\frac{1}{2}\sigma^2 \frac{\partial^2 u}{\partial x^2}(t,x) + \mu\frac{\partial u}{\partial x}(t,x) = -\frac{\partial u}{\partial t}(t,x), \quad \mu = r - \frac{\sigma^2}{2}, \quad u(T,x) = f(e^x)$$

Put example

- $f(e^x) = \max(K - e^x, 0)$ where K is the strike
- The two limit conditions will be set at $x_{min} = \ln(K/2)$ and $x_{max} = \ln(2K)$ assuming heuristically that for all $t \in [0, T]$

$$u(t, x_{min}) = \texttt{pmin} = K/2 \quad \& \quad u(t, x_{max}) = \texttt{pmax} = 0 \tag{4}$$

PDE discretization

- σ takes its value in $[0.1, 0.5]$
- The volatility discretization involves $\texttt{NB} = 64$ cells
- The space discretization involves $\texttt{NTPB} = 256$ cells
- The time discretization of $[0, T]$ involves $\texttt{N} = 10000$ time steps

EUMaster4HPC

PDE Crank-Nicolson $u_{i,j} = u(t_i, x_j)$

$$q_u u_{i,j+1} + q_m u_{i,j} + q_d u_{i,j-1} = p_u u_{i+1,j+1} + p_m u_{i+1,j} + p_d u_{i+1,j-1},$$

$$p_u = \frac{\sigma^2 \Delta t}{4\Delta x^2} + \frac{\mu \Delta t}{4\Delta x}, \quad p_m = 1 - \frac{\sigma^2 \Delta t}{2\Delta x^2}, \quad p_d = \frac{\sigma^2 \Delta t}{4\Delta x^2} - \frac{\mu \Delta t}{4\Delta x} \tag{5}$$

$$q_u = -\frac{\sigma^2 \Delta t}{4\Delta x^2} - \frac{\mu \Delta t}{4\Delta x}, \quad q_m = 1 + \frac{\sigma^2 \Delta t}{2\Delta x^2}, \quad q_d = -\frac{\sigma^2 \Delta t}{4\Delta x^2} + \frac{\mu \Delta t}{4\Delta x}$$

Tridiagonal system

$$\begin{pmatrix} d_1 & c_1 & & & \\ a_2 & d_2 & c_2 & & 0 \\ & a_3 & d_3 & \ddots & \\ & & \ddots & \ddots & \ddots \\ & 0 & & \ddots & \ddots & c_{n-1} \\ & & & & a_n & d_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \tag{6}$$

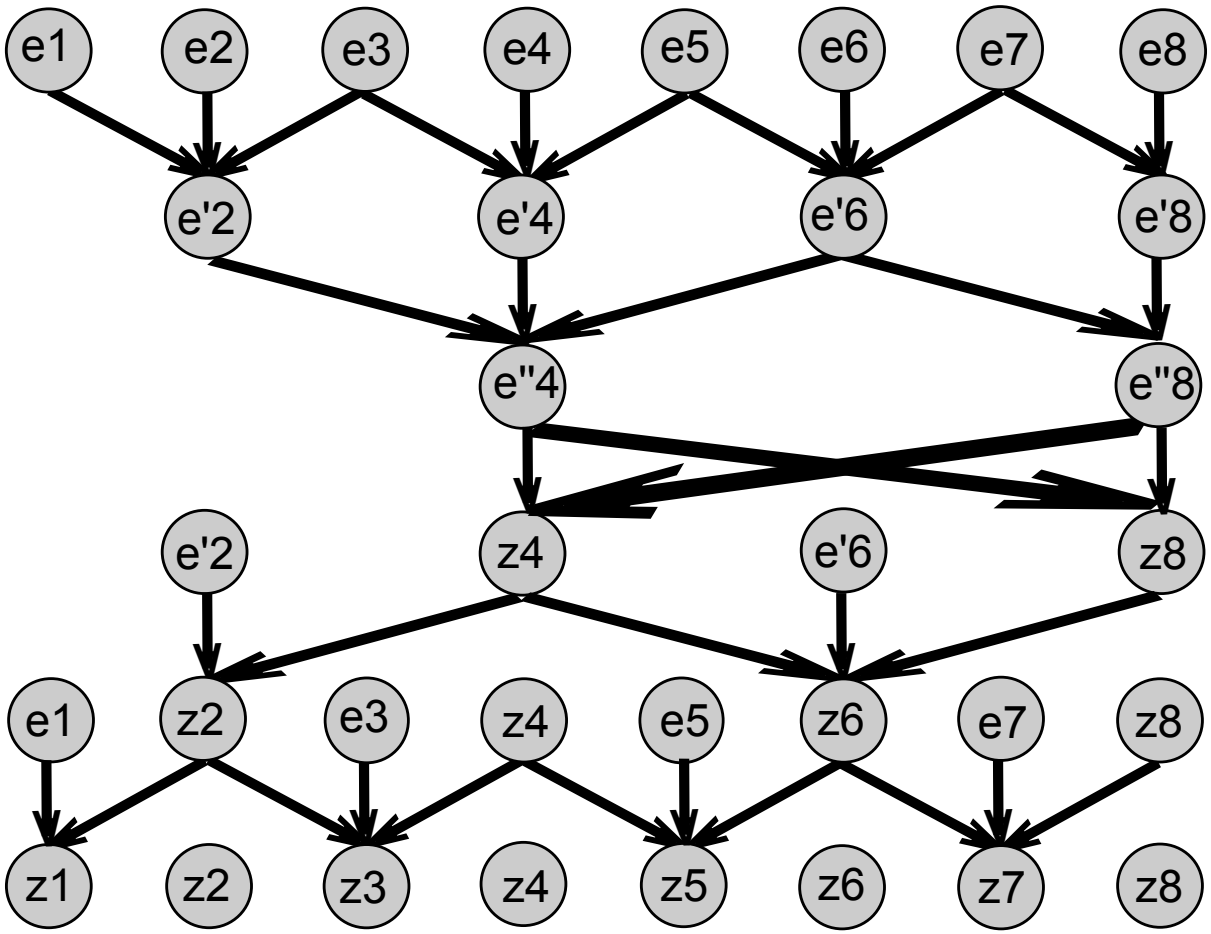
Reference for PCR

L. A. Abbas-Turki and Stef Graillat, "Resolving small random symmetric linear systems on graphics processing units", The Journal of Supercomputing 73(4), pp. 1360–1386, 2017

EUMaster4HPC

From CR: Shared occupation $4n$ and complexity $O(nlog_2(n))$

CR when $n = 8$



Step 1: Forward reduction to a 4-unknown system involving z2, z4, z6 and z8

Step 2: Forward reduction to a 2-unknown system involving z4 and z8

Step 3: Solve 2-unknown system

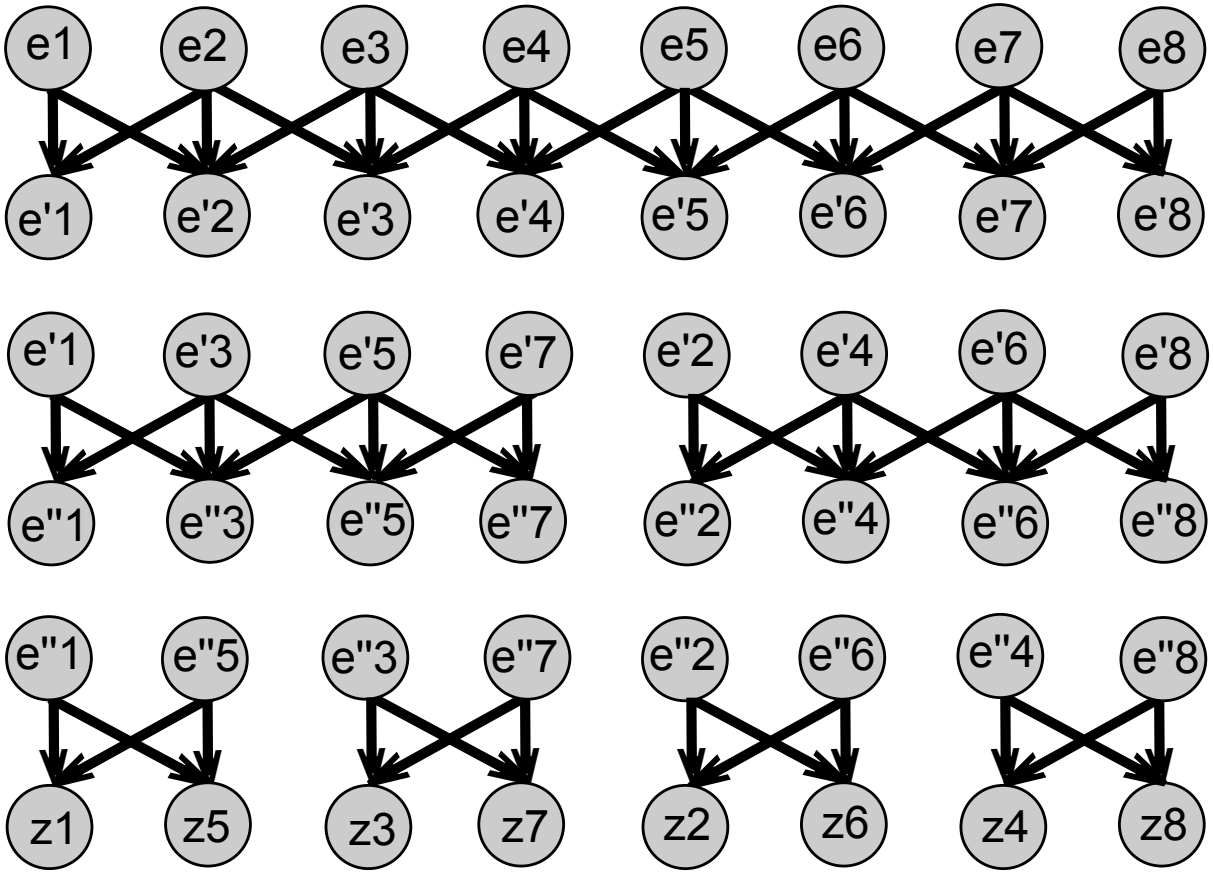
Step 4: Backward substitution to solve the rest 2 unknowns

Step 5: Backward substitution to solve the rest 4 unknowns

EUMaster4HPC

To a new version of PCR: Shared occupation $5n$ and complexity $O(nlog_2(n))$

PCR when $n = 8$



Step 1: Reduced to 2 systems of 4 unknowns

Step 2: Reduced to 4 systems of 2 unknowns

Step 3: Solve

EUMaster4HPC

$$\begin{pmatrix} d_1 & c_1 & & & & & \\ a_2 & d_2 & c_2 & & & & \\ & a_3 & d_3 & c_3 & & & \\ & & a_4 & d_4 & c_4 & & \\ & & & a_5 & d_5 & c_5 & \\ & & & & a_6 & d_6 & c_6 \\ & & & & & a_7 & d_7 \end{pmatrix}$$

$(R)\longrightarrow$

$$\begin{pmatrix} d'_1 & 0 & c'_1 & & & & \\ 0 & d'_2 & 0 & c'_2 & & & \\ a'_3 & 0 & d'_3 & 0 & c'_3 & & \\ & a'_4 & 0 & d'_4 & 0 & c'_4 & \\ & & a'_5 & 0 & d'_5 & 0 & c'_5 \\ & & & a'_6 & 0 & d'_6 & 0 \\ & & & & a'_7 & 0 & d'_7 \end{pmatrix}$$

$(P)\longrightarrow$

$$\begin{pmatrix} d'_1 & c'_1 & & & & & \\ a'_3 & d'_3 & c'_3 & & & & \\ & a'_5 & d'_5 & c'_5 & & & \\ & & a'_7 & d'_7 & 0 & & \\ & & & 0 & d'_2 & c'_2 & \\ & & & & a'_4 & d'_4 & c'_4 \\ & & & & & a'_6 & d'_6 \end{pmatrix}$$

$(R)\longrightarrow$

$$\begin{pmatrix} d''_1 & 0 & c''_1 & & & & \\ 0 & d''_3 & 0 & c''_3 & & & \\ a''_5 & 0 & d''_5 & 0 & & & \\ & a''_7 & 0 & d''_7 & 0 & & \\ & & & 0 & d''_2 & 0 & c''_2 \\ & & & & 0 & d''_4 & 0 \\ & & & & a''_6 & 0 & d''_6 \end{pmatrix}$$

$(P)\longrightarrow$

$$\begin{pmatrix} d''_1 & c''_1 & & & & & \\ a''_5 & d''_5 & 0 & & & & \\ & 0 & d''_3 & c''_3 & & & \\ & & a''_7 & d''_7 & 0 & & \\ & & & 0 & d''_2 & c''_2 & \\ & & & & a''_6 & d''_6 & 0 \\ & & & & & 0 & d''_4 \end{pmatrix}$$

EUMaster4HPC