GPU Computing and Accelerators: Part IV

Definition (Stream)

Streams are a mechanism that introduces an additional level of parallelism into the CUDA framework. While the basic setup, we have seen until here, is SIMD or more precisely SIMT, using streams one can have the GPU do different things at the same time. Streams are not as flexible and "general purpose" as tasks on the host CPU, though.

The basic power of streams is to have memory transfers and computational operations overlap in an asynchronous way. Note, however, that not all CUDA enabled devices support overlapping these operations. On top of that, not all CUDA enabled devices that do support the overlapping execution do so in the same way.

Compute Unified Device Architecture (CUDA)

Page-Locked Memory on the Host

Asynchronous data transfers in CUDA are not only performed without synchronization to the actual computation, they are also intended to interact with the computation as little as possible. Especially, they should not interrupt the CPU from performing useful work in the program. They are therefore set up to use direct memory access (DMA) circumventing CPU interaction.

However, in order to do this, we need to use a special portion of host memory, that is guaranteed to stay in place during the operation. The default portion of host memory that we allocate using `malloc()` is paged memory. It can be anywhere in the virtual memory of the host and is allowed to move around, e.g., to get swapped to disk when more space is required.

Definition (page-locked memory)

Page-locked memory is a portion of memory that is guaranteed to keep its position in the virtual memory. It is not available for any kind of paging operations, such as swapping. Therefore, it is sometimes also called pinned memory.

Advantages of pinned memory:

- can be used for DMA safely
- transfer speeds can be up to $2\times$ faster than to/from pageable memory

Disadvantages:

- memory fragmentation increases and thus the usability deteriorates.

Compute Unified Device Architecture (CUDA)

Streams and Compute Capabilities

Over the years NVIDIA® has changed the way things are implemented. This is not only regarding the API in the CUDA toolkit, but also the underlying device hardware. The very first CUDA enabled devices could not overlap transfers and executions at all. Then some devices used separate engines for copy and kernel executions. Modern hardware usually has even two engines for performing transfers in direction to the host and to the device separately. Basically we can classify the devices as follows:

Comp. Capab.	Properties
1.0	No overlap
1.1-	1 copy engine and 1 kernel execution engine
2.x-	1 kernel execution engine, 1 copy to host engine and 1 copy to device engine
3.5-	eliminates the differences in asynchronous execution

Table: classification of CUDA enabled devices with respect to the ability of overlapping memory transfers and computations.

How can i know what my device can do?

The `cudaDeviceProp` structure can be used to find out whether a device supports overlapped operation and how many execution engines are available. The important members are

- `int deviceOverlap` indicating the availability of overlapped operations
- `int asyncEngineCount` storing the number of asynchronous execution engines available.

The important information, which type of asynchronous execution model is implemented in the hardware can thus be fetched with the `cudaGetDeviceProperties()` function.

Compute Unified Device Architecture (CUDA)

An Introductory Asynchronous Transfer Example

We are following [5] What we want to do is

- copy data to the device
- perform some task (kernel) on it
- get the result back to the host

Example

The the critical portion of the code would look like

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>>(d_a)
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

according to what we have learned until now. This is regarding the default execution stream.

[5] https://developer.nvidia.com/content/how-overlap-data-transfers-cuda-cc

Example (Creation and Destruction of Streams)

Consider we have the two variables

```
cudaStream_t stream1;
cudaError_t result;
```

Then we can create a new stream using

```
result = cudaStreamCreate(&stream1)
```

and later get rid of it via

```
result = cudaStreamDestroy(stream1)
```

Example (Memory transfers)

Once we have acquired a new stream we have to tell the asynchronous copy routines to use it. The basic command `cudaMemcpyAsync()` takes the same arguments as `cudaMemcpy`. Only, it has an additional argument specifying the stream to use:

```
result = cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1)
```

Example (Kernel Execution)

We need to use the extended launch size specification here:

```
<<< block distr., thread distr., dynamic memory per block, associated
    stream >>>
```

The third argument can be used to allocate additional dynamic shared memory per block. We will use 0 here.

```
kernel<<<1,N,0,stream1>>>(d_a)
```

The influence of the number of engines (especially for copying data) is best displayed in a simple example.

Consider we have a group of streams cooperating on `kernel()`. Think of a situation where splitting the problem data into chunks is necessary to fit the data into the device memory. We basically have two ways to implement the cooperation,

1. loop over the entire copy-work-copy block
2. loop over the work and copies separately

Note that the asynchronous copy acts different on the control flow than the `cudaMemcpy()`. While in the default stream, using `cudaMemcpy()`, we can rely on the fact that as soon as the command returns, all data has been transferred, in the case of `cudaMemcpyAsync()` it does not even guarantee that the copy operation has started at all. It will only have scheduled the operation in a first in first out (FIFO) list of pending operations on the corresponding asynchronous execution engine.

Example (Asynchronous Execution Version 1)

Looping over the entire block of copy-work-copy operations is described by the following code fragment

```
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes, stream[i]);
  kernel<<<>>>(d_a, offset);
  cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, stream[i]);
}
```

Example (Asynchronous Execution Version 2)

Looping over the single tasks in contrast looks like

```
for (int i = 0; i < nStreams; ++i) {
 int offset = i * streamSize;
 cudaMemcpyAsync(&d_a[offset], &a[offset],
                 streamBytes, cudaMemcpyHostToDevice, stream[i]);
}

for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  kernel<<<>>>(d_a, offset);
}

for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&a[offset], &d_a[offset],
                  streamBytes, cudaMemcpyDeviceToHost, stream[i]);
}
```

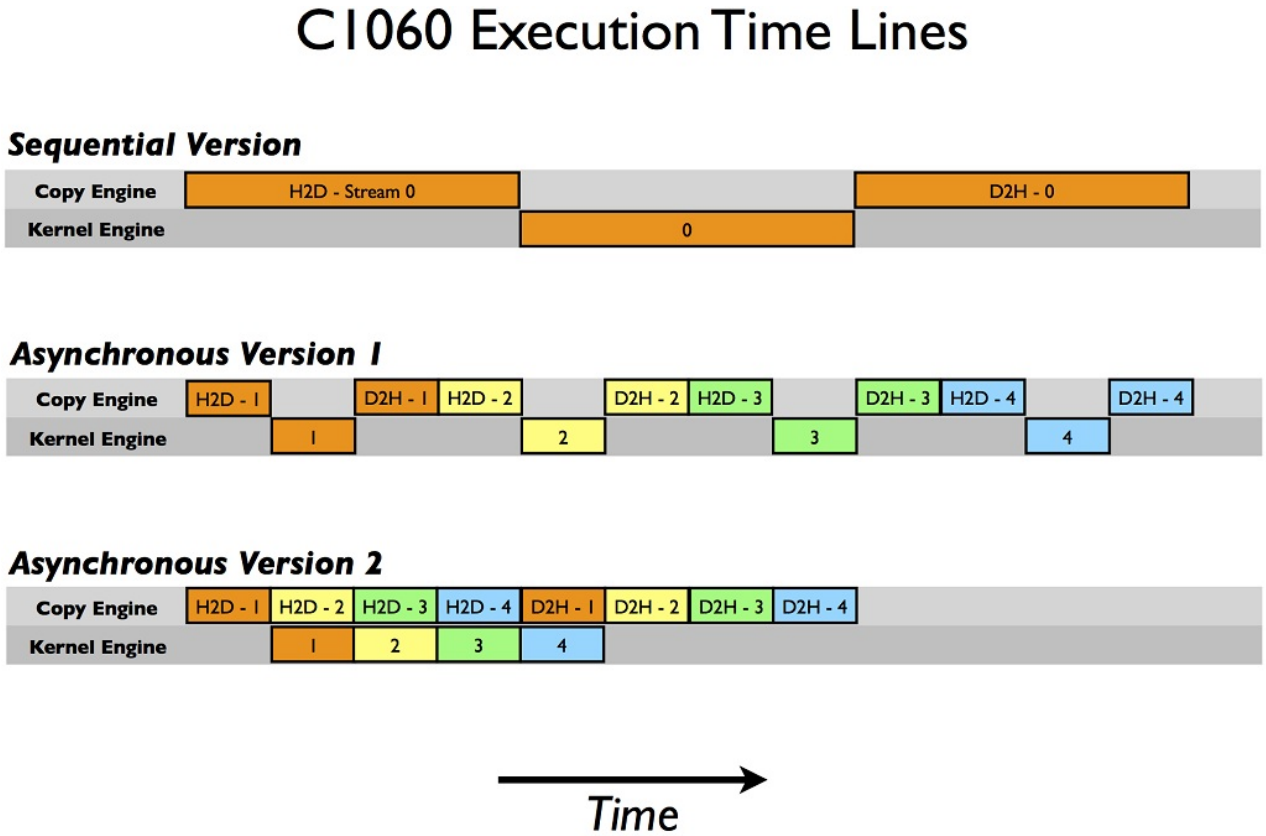


Figure: Execution time line on a device with a single copy engine.

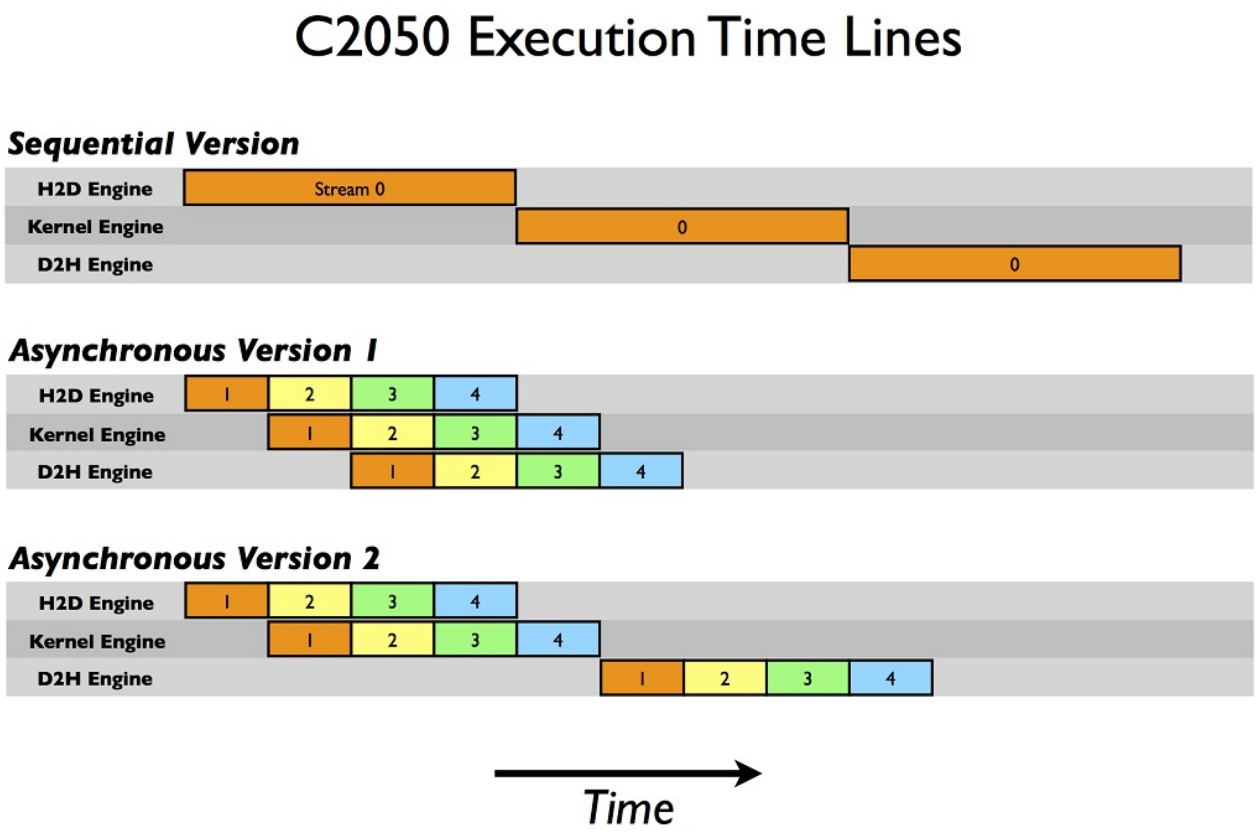


Figure: Execution time line on a device with separate copy engines for device to host (D2H) and host to device (H2D) operations.

Compute Unified Device Architecture (CUDA)

Interoperability with Graphics

Using the same GPU for computations and graphical display of results is possible. See, e.g. CUDA by Example (Chapter 8), or CUDA C Programming Guide.

Compute Unified Device Architecture (CUDA)

Interoperability with Graphics

Using the same GPU for computations and graphical display of results is possible. See, e.g. CUDA by Example (Chapter 8), or CUDA C Programming Guide.

Usage of Multiple GPUs

Usage of multiple GPUs in a single program requires the concepts of zero-copy host memory, and portable pinned memory. An introduction can be found in CUDA by Example (Chapter 11).

Chapter 4

GPU Computing and Accelerators: Part V

Open Computing Language (OpenCL)

Main Message

The abstraction for the programming and hardware models are very similar to the CUDA concepts. Mainly OpenCL delivers slightly more flexible implementations due to vendor independence and uses slightly different vocabulary for the single ingredients of the concept.

CUDA	OpenCL
thread	(Work) item
block	(Work) group
streaming multiprocessor	compute unit
(CUDA) processor	processing unit

Table: A short CUDA to OpenCL dictionary

Hybrid CPU-GPU Linear System Solvers

The block outer product LU decomposition revisited

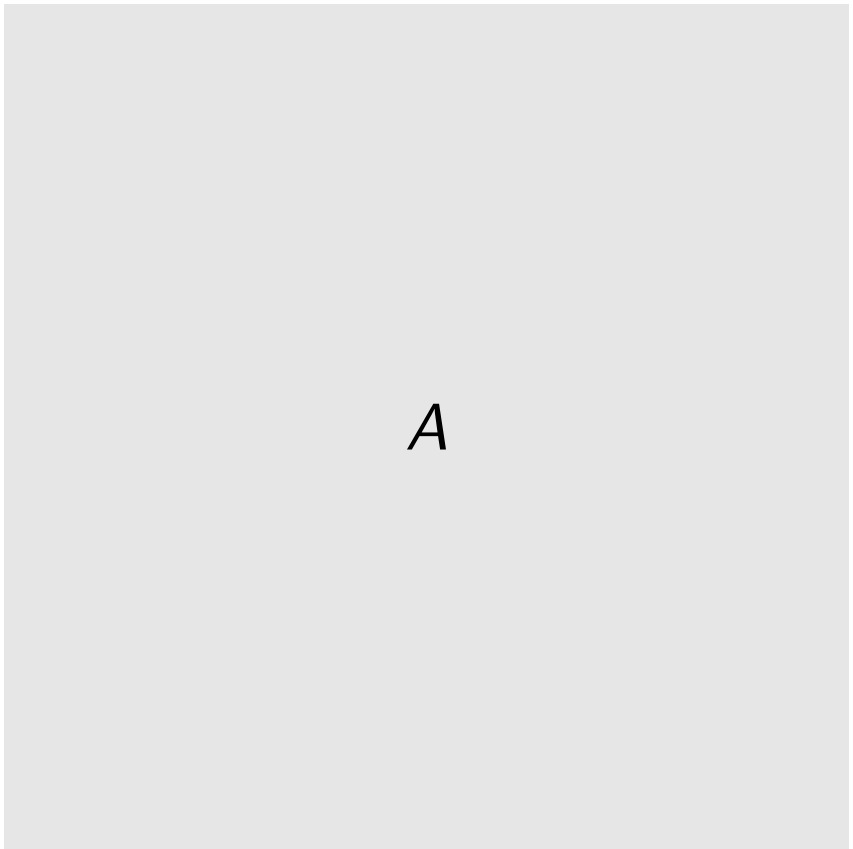
Algorithm 6: Gaussian elimination – Block outer product formulation

Input: $A \in \mathbb{R}^{n \times n}$ allowing LU decomposition, r prescribed block size
Output: $A = LU$ with L, U stored in A

1 $k = 1$;
2 **while** $k \leq n$ **do**
3 $\quad \ell = \min(n, k + r - 1)$;
4 $\quad$ Compute $A(k : \ell, k : \ell) = \tilde{L}\tilde{U}$ via Algorithm 7;
5 $\quad$ Solve $\tilde{L}Z = A(k : \ell, \ell + 1 : n)$ and store Z in A;
6 $\quad$ Solve $W\tilde{U} = A(\ell + 1 : n, k : \ell)$ and store W in A;
7 $\quad$ Perform the rank-r update:
$\quad A(\ell + 1 : n, \ell + 1 : n) = A(\ell + 1 : n, \ell + 1 : n) - WZ$;
8 $\quad k = \ell + 1$;

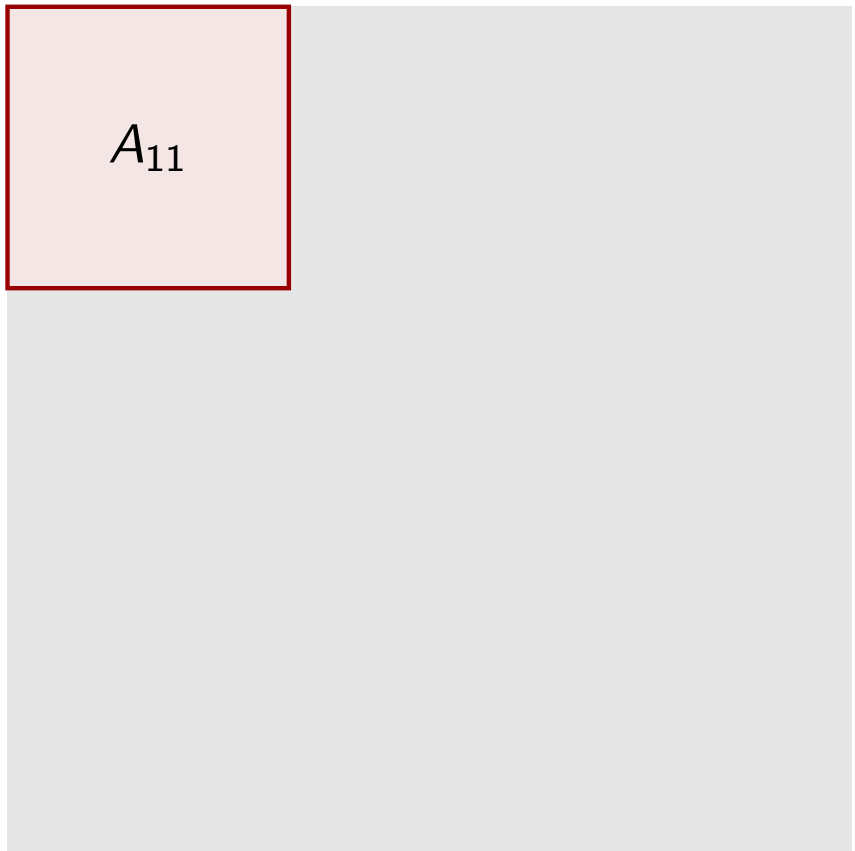
Hybrid CPU-GPU Linear System Solvers

The block outer product LU decomposition revisited



Hybrid CPU-GPU Linear System Solvers

The block outer product LU decomposition revisited

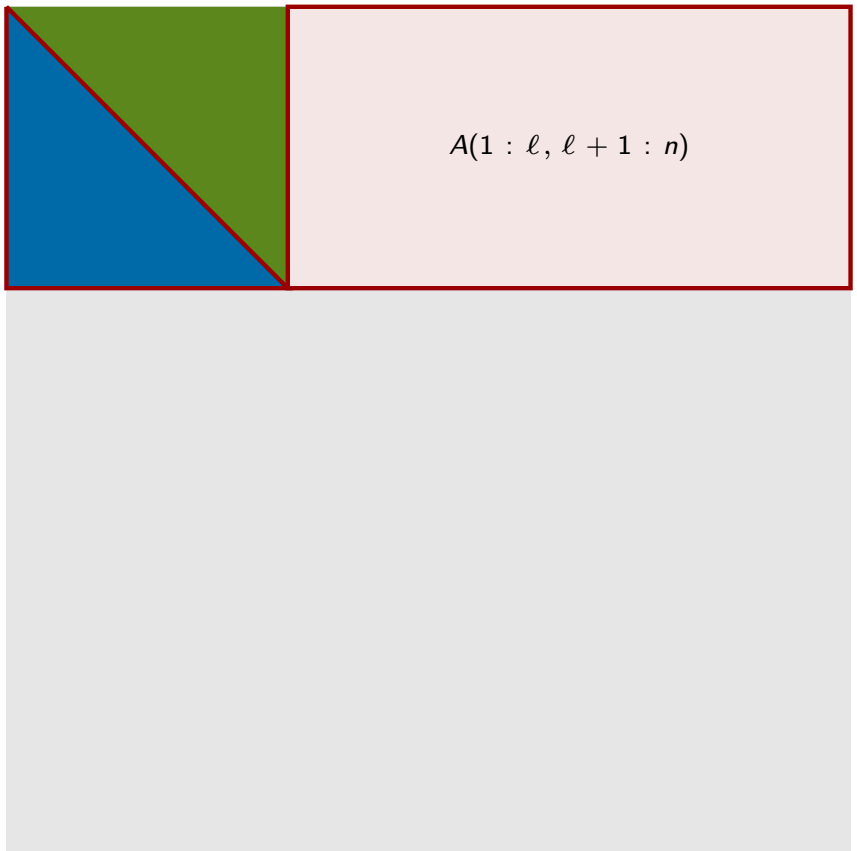


Hybrid CPU-GPU Linear System Solvers

The block outer product LU decomposition revisited

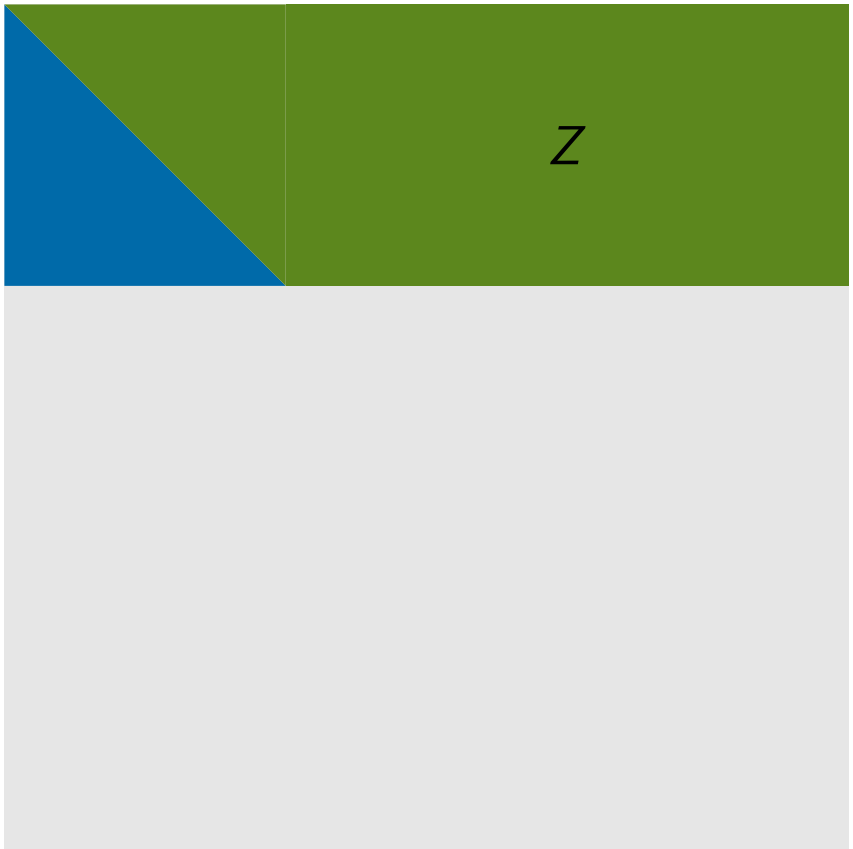
Hybrid CPU-GPU Linear System Solvers

The block outer product LU decomposition revisited



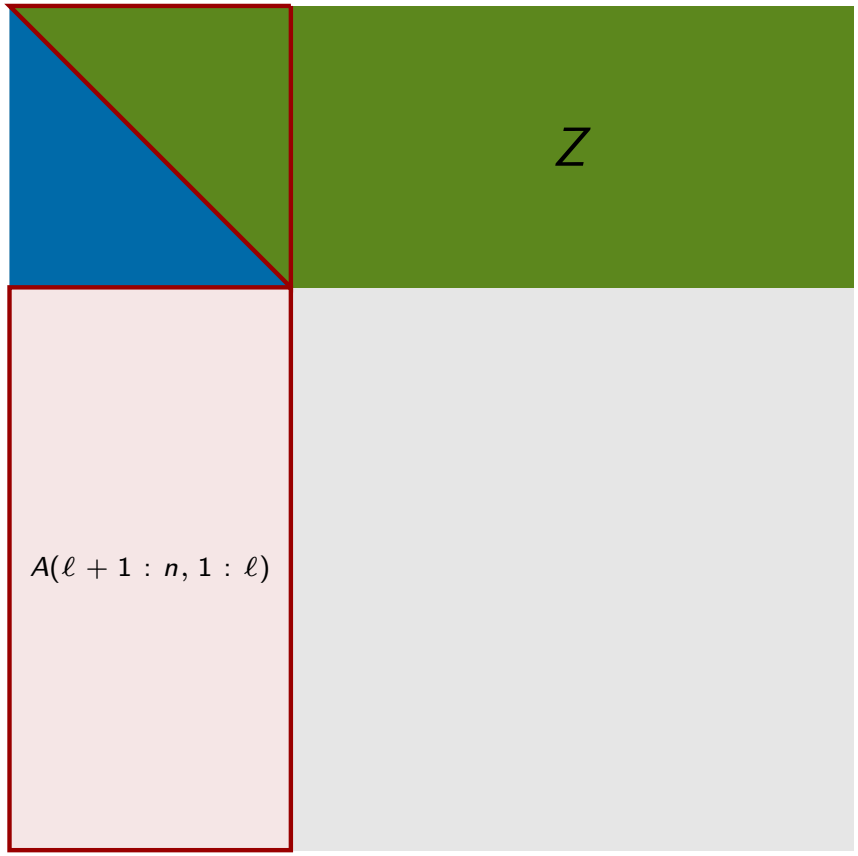
Hybrid CPU-GPU Linear System Solvers

The block outer product LU decomposition revisited



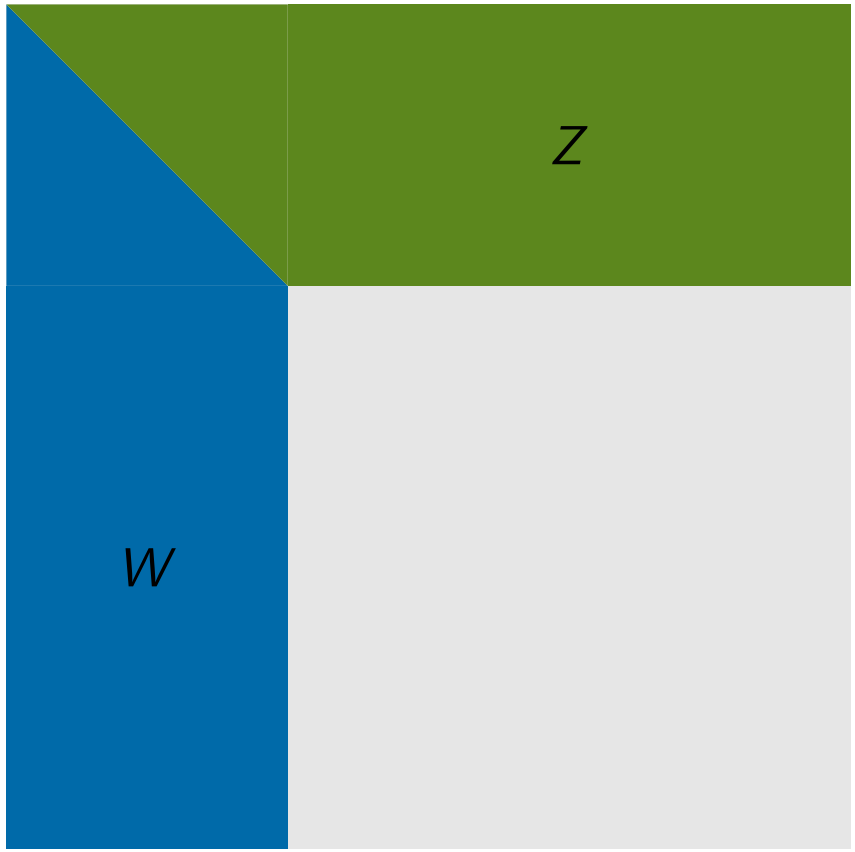
Hybrid CPU-GPU Linear System Solvers

The block outer product LU decomposition revisited



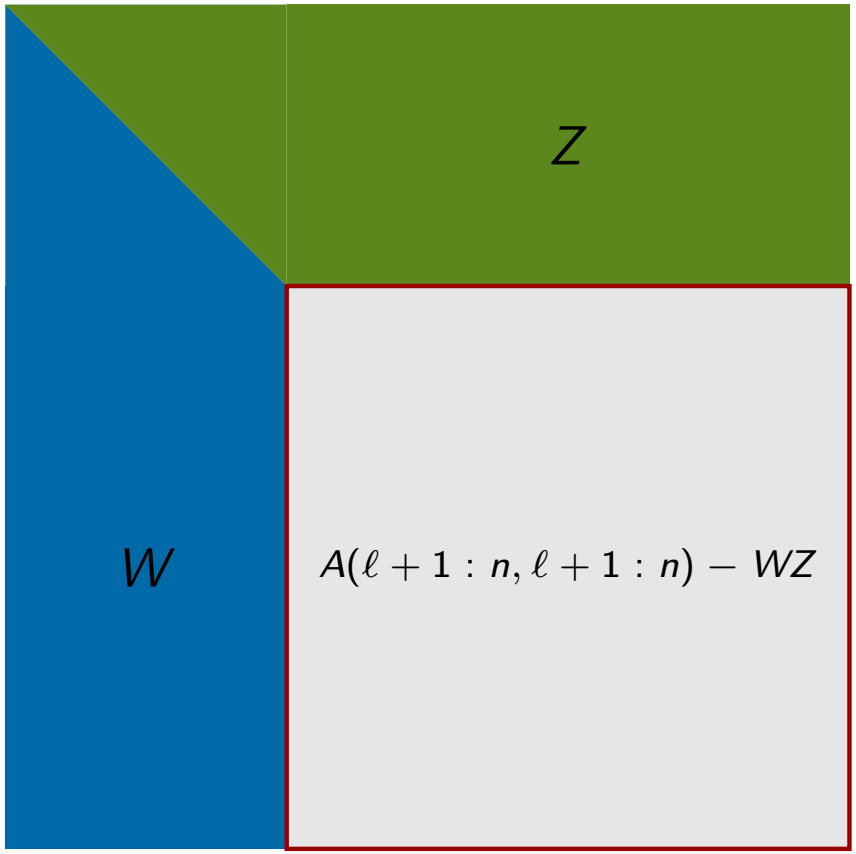
Hybrid CPU-GPU Linear System Solvers

The block outer product LU decomposition revisited



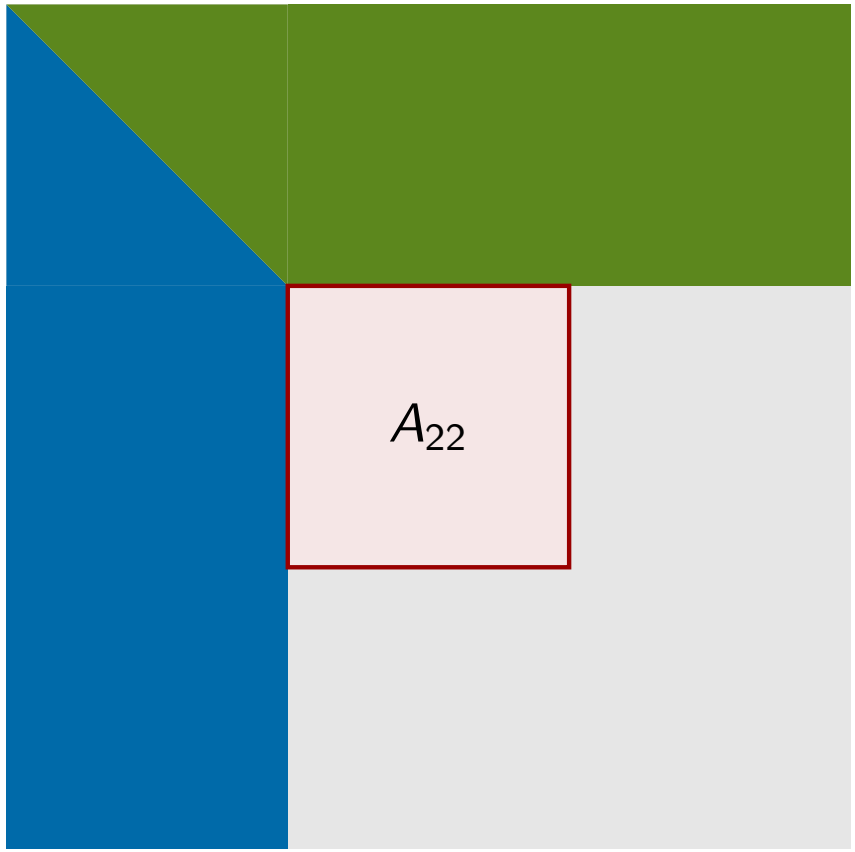
Hybrid CPU-GPU Linear System Solvers

The block outer product LU decomposition revisited



Hybrid CPU-GPU Linear System Solvers

The block outer product LU decomposition revisited



Hybrid CPU-GPU Linear System Solvers

The block outer product LU decomposition revisited

Hybrid CPU-GPU Linear System Solvers

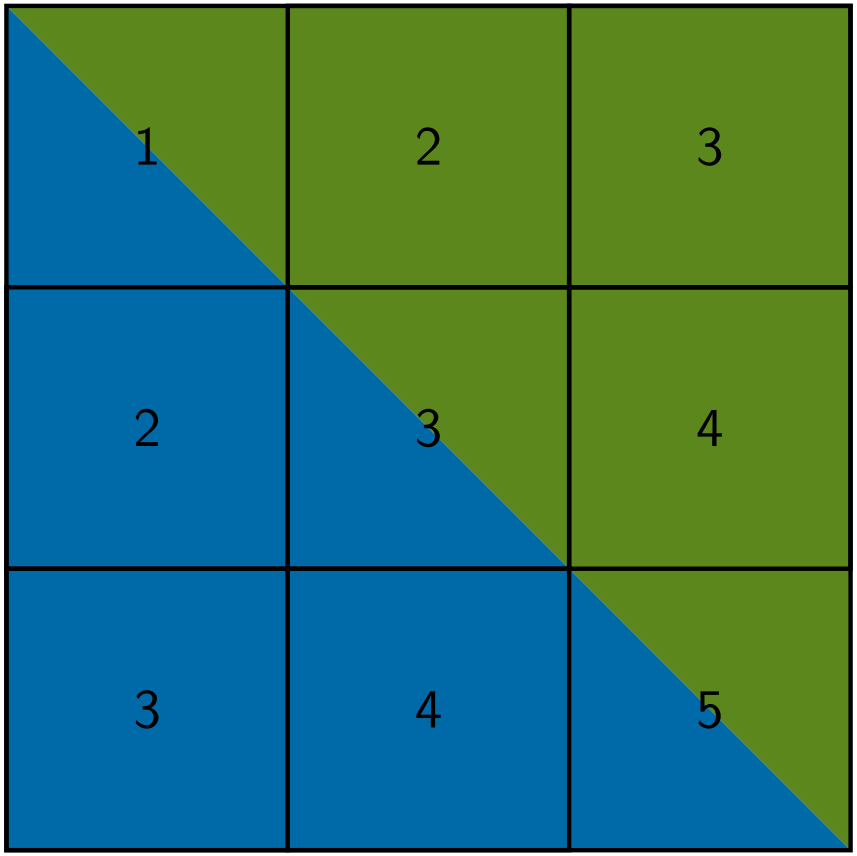
The block outer product LU decomposition revisited

Hybrid CPU-GPU Linear System Solvers

The block outer product LU decomposition revisited

Hybrid CPU-GPU Linear System Solvers

The block outer product LU decomposition revisited



Hybrid CPU-GPU Linear System Solvers

The block outer product LU decomposition revisited

The central question for the hybrid CPU/GPU version of the algorithm now is where to execute the single steps of the algorithm compared to the DAG scheduled version.

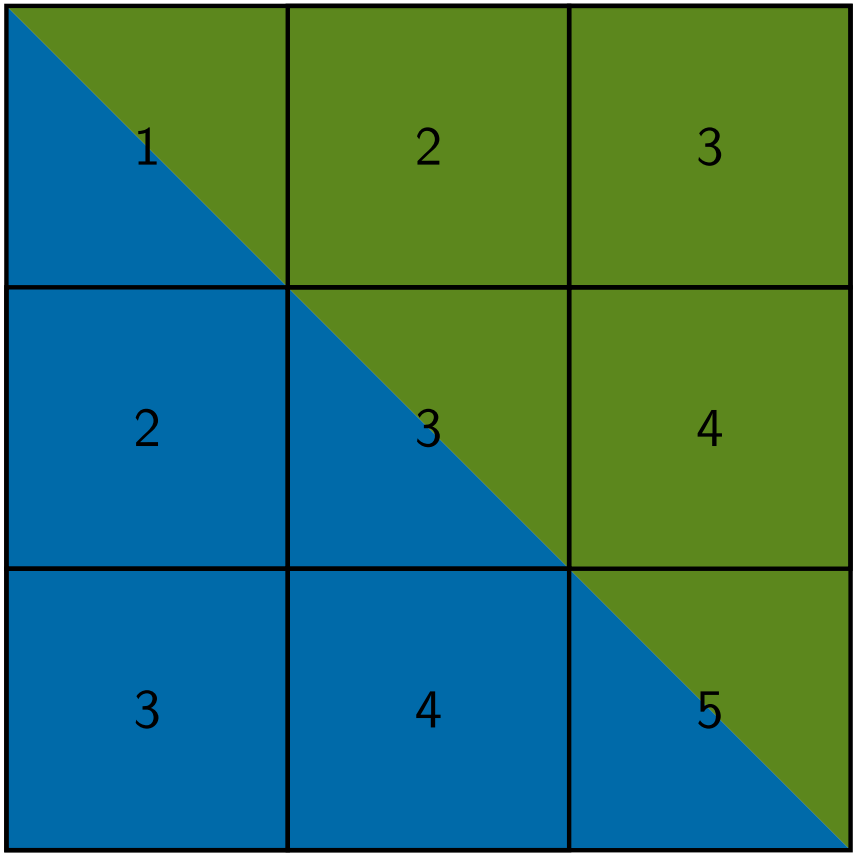
Requirements

- Keep data transfers between host and device limited
- optimize usage of both host and device features
- assume that the entire matrix fits into the device memory.

The assumption on the matrix size may be loosened but will then lead to a completely different algorithm.

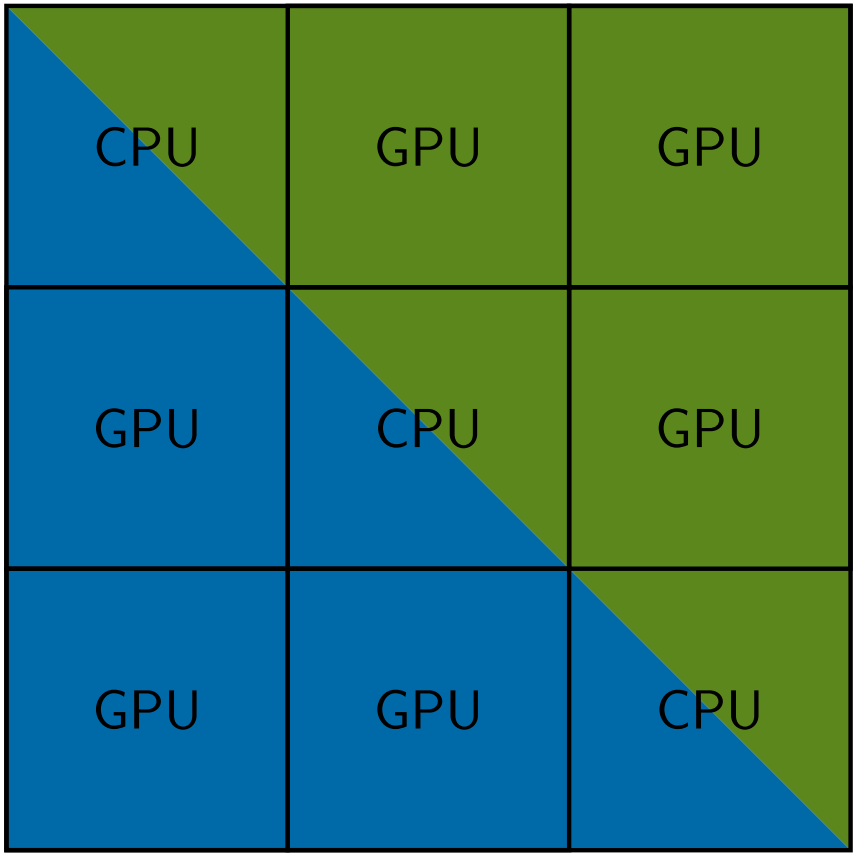
Hybrid CPU-GPU Linear System Solvers

The block outer product LU decomposition revisited



Hybrid CPU-GPU Linear System Solvers

The block outer product LU decomposition revisited



Hybrid CPU-GPU Linear System Solvers

The block outer product LU decomposition revisited

In each outer iteration step perform the leading $r \times r$ blocks LU decomposition

Hybrid CPU-GPU Linear System Solvers

Iterative Linear System Solvers

Algorithm 6: Conjugate Gradient Method

Input: $A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n, x_0 \in \mathbb{R}^n$
Output: $x = A^{-1}b$

$p_0 = r_0 = b - Ax_0,\ \alpha_0 = \|r_0\|_2^2$;
for $m = 0, \ldots, n-1$ **do**
  if $\alpha_m \neq 0$ **then**
    $v_m = Ap_m$;
    $\lambda_m = \frac{\alpha_m}{(v_m, p_m)}$;
    $x_{m+1} = x_m + \lambda_m p_m$;
    $r_{m+1} = r_m - \lambda_m v_m$;
    $\alpha_{m+1} = \|r_{m+1}\|_2^2$;
    $p_{m+1} = r_{m+1} + \frac{\alpha_{m+1}}{\alpha_m} p_m$;
  else
    STOP;

Hybrid CPU-GPU Linear System Solvers

Iterative Linear System Solvers

There are mainly two observations we can draw from the algorithm.

1. The single steps need to be executed mainly sequentially
2. basically all operations are vector operations.

There is not much to distribute between host and device. To exploit the devices vector features all operations should be executed on the device. In case the matrix can not be stored in device memory completely it may be beneficial to use streams to split the operation into chunks that can be stored and operate on those streams in a round robin fashion.

Hybrid CPU-GPU Linear System Solvers

Sparse Iterative Eigenvalue Approximation

Basic Idea

- Very similar to iterative linear solvers based on Krylov subspaces.
- Main ingredient is to use the basis of the subspace to project the eigenvalue problem to a much smaller space and solve it with dense methods there, i.e. $A \in \mathbb{R}^{n \times n}$ large and sparse $U \in \mathbb{R}^{m \times n}$, $m \ll n$ orthogonal, then

$$\underbrace{UAU^T}_{m \times m} x = \lambda x$$

  is an m-dimensional dense eigenproblem.

Here one can offload the solution of the small eigenvalue problem to the host, while the device keeps extending the basis further. The host can then decide whether the approximation is good enough, or the extension is required and the computation needs to continue.

Relevant Software and Libraries

The CUDA Related Libraries

- **CUDA Math** provides basically all math functions in `math.h` as device functions.
- **CUBLAS** the CUDA deice based implementation of BLAS
- **CUFFT** CUDA based Fast Fourier Transforms, i.e., divide and conquer based computation of Fourier transforms of complex and real valued data sets.
- **CURAND** The CURAND library provides facilities that focus on the simple and efficient generation of high-quality pseudorandom and quasirandom numbers.
- **CUSPARSE** Vector-vector and matrix-vector operations where at least one participant is sparse.
- **Thurst** A C++ template library based on the Standard Template library (STL) for minimal effort implementation of parallel programs.

Relevant Software and Libraries

Matrix Algebra on GPU and Multicore Architectures (MAGMA)[a]

[a] http://icl.cs.utk.edu/magma/index.html

"The MAGMA project aims to develop a dense linear algebra library similar to LAPACK but for heterogeneous/hybrid architectures, starting with current "Multicore+GPU" systems.
The MAGMA research is based on the idea that, to address the complex challenges of the emerging hybrid environments, optimal software solutions will themselves have to hybridize, combining the strengths of different algorithms within a single framework. Building on this idea, we aim to design linear algebra algorithms and frameworks for hybrid manycore and GPU systems that can enable applications to fully exploit the power that each of the hybrid components offers."

Relevant Software and Libraries

Formal Linear Algebra Methodology Environment (FLAME)[a]

[a] http://www.cs.utexas.edu/~flame/web/

"The objective of the FLAME project is to transform the development of dense linear algebra libraries from an art reserved for experts to a science that can be understood by novice and expert alike. Rather than being only a library, the project encompasses a new notation for expressing algorithms, a methodology for systematic derivation of algorithms, Application Program Interfaces (APIs) for representing the algorithms in code, and tools for mechanical derivation, implementation and analysis of algorithms and implementations."

Relevant Software and Libraries

CUSP[a]

[a] https://github.com/cusplibrary

"Cusp is a library for sparse linear algebra and graph computations on CUDA. Cusp provides a flexible, high-level interface for manipulating sparse matrices and solving sparse linear systems. Get Started with Cusp today!"

Relevant Software and Libraries

CUSP[a]

[a] https://github.com/cusplibrary

"Cusp is a library for sparse linear algebra and graph computations on CUDA. Cusp provides a flexible, high-level interface for manipulating sparse matrices and solving sparse linear systems. Get Started with Cusp today!"

Matrix formats:

- Coordinate (COO)
- Compressed Sparse Row (CSR)
- Diagonal (DIA)
- ELL (ELL)
- Hybrid (HYB)

Relevant Software and Libraries

CUSP[a]

[a] https://github.com/cusplibrary

"Cusp is a library for sparse linear algebra and graph computations on CUDA. Cusp provides a flexible, high-level interface for manipulating sparse matrices and solving sparse linear systems. Get Started with Cusp today!"

More Features:

- Format conversion
- Dense Arrays
- File I/O (Matrix Market format)

Relevant Software and Libraries

CUSP[a]

[a] https://github.com/cusplibrary

"Cusp is a library for sparse linear algebra and graph computations on CUDA. Cusp provides a flexible, high-level interface for manipulating sparse matrices and solving sparse linear systems. Get Started with Cusp today!"

Supported Iterative Solvers:

- Conjugate-Gradient (CG)
- Biconjugate Gradient (BiCG)
- Biconjugate Gradient Stabilized (BiCGstab)
- Generalized Minimum Residual (GMRES)
- Multi-mass Conjugate-Gradient (CG-M)
- Multi-mass Biconjugate Gradient stabilized (BiCGstab-M)

Relevant Software and Libraries

CUSP[a]

[a] https://github.com/cusplibrary

"Cusp is a library for sparse linear algebra and graph computations on CUDA. Cusp provides a flexible, high-level interface for manipulating sparse matrices and solving sparse linear systems. Get Started with Cusp today!"

Preconditioners:

- Algebraic Multigrid (AMG) based on Smoothed Aggregation
- Approximate Inverse (AINV)
- Diagonal

Relevant Software and Libraries

CULA tools[a]

[a] http://www.culatools.com

"CULA is a set of GPU-accelerated linear algebra libraries utilizing the NVIDIA CUDA parallel computing architecture to dramatically improve the computation speed of sophisticated mathematics."

They have separate packages for sparse and dense operation. The libraries are however commercial.

Besides those, there are many scientific computing packages that support GPU operations in one way or the other. Also python has packages for both CUDA (pyCUDA) and OpenCL (pyOpenCL) and MATLAB supports (basically dense only) operation on CUDA devices.