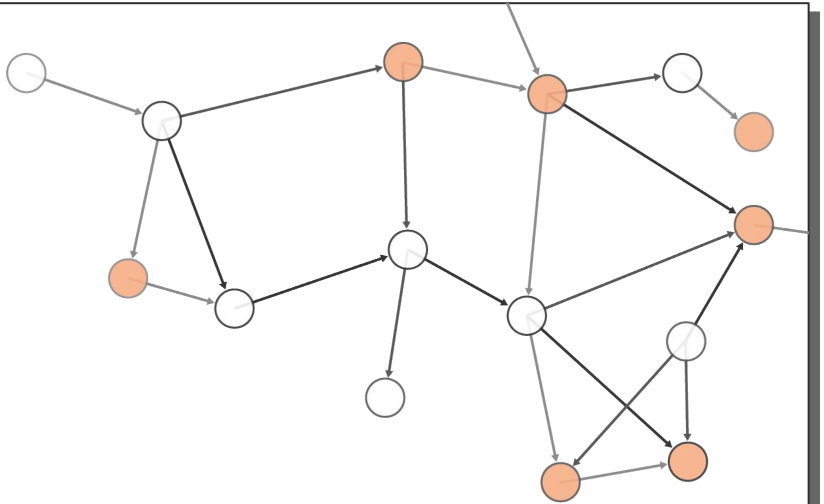


# DAG-based Scheduling with Resource Sharing for Multi-task Applications in a Polyglot GPU Runtime


Alberto Parravicini, Politecnico di Milano, alberto.parravicini@polimi.it

Arnaud Delamare, Oracle Labs

Marco Arnaboldi, Oracle Labs

Marco D. Santambrogio, Politecnico di Milano


2021-05-18

POLITECNICO MILANO 1863

POLITECNICO MILANO 1863 NECST laboratory

# Improving performance in multi-kernel GPU computations

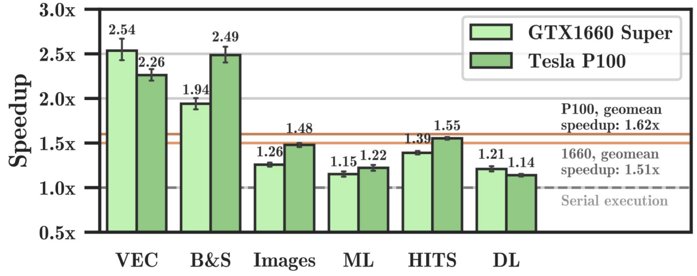


GPUs are great for **parallel computing**

- Deep Learning, Image processing, Graph analytics, etc.

But multi-kernel applications offer more opportunities for asynchronous computations

1. **Run concurrent GPU computations (*space-sharing*)**
2. **Run GPU computations concurrently to CPU**
3. **Overlap data-transfer with computations**




Asynchronous execution provides an average of 62% speedup on a Tesla P100

# Improving performance in multi-kernel GPU computations

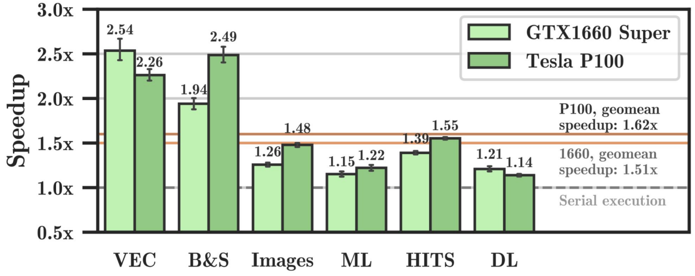


GPUs are great for **parallel computing**

- Deep Learning, Image processing, Graph analytics, etc.

But multi-kernel applications offer more opportunities for asynchronous computations

1. **Run concurrent GPU computations (*space-sharing*)**
2. **Run GPU computations concurrently to CPU**
3. **Overlap data-transfer with computations**




**Extracting full performance in multi-kernel computations is hard**

- Synchronization events and data-movement must be **hand-optimized**
- Full CUDA API is only available to C/C++

# Achieving peak performance in multi-kernel, automatically



**We want to provide fully transparent & automatic GPU scheduling**

- A new scheduler that provides GPU space-sharing, CPU/GPU overlap, data-transfer/computation overlap
- High-level abstraction: support high-level languages (Python, R, JavaScript, Scala, etc.) through the **GrCUDA** API, **without changing it**
- Same performance as low-level APIs: CUDA Graphs, hand-optimized CUDA events in C++

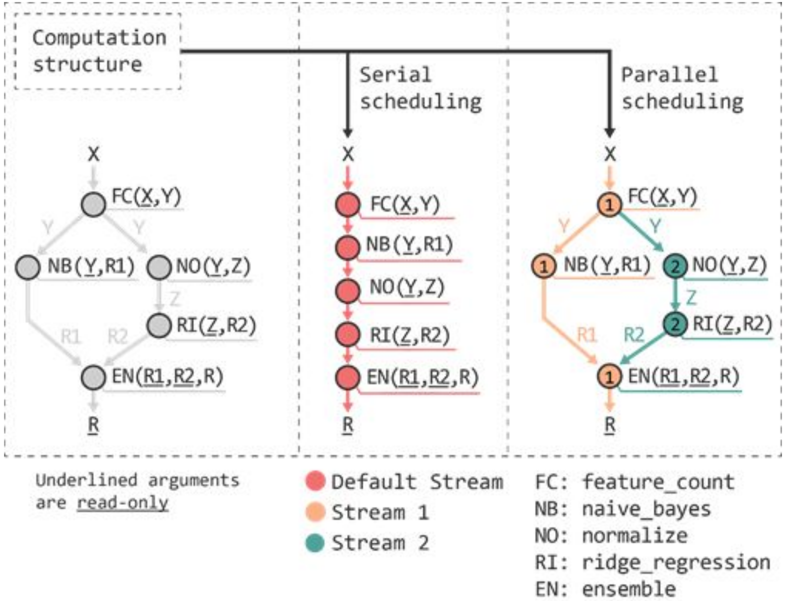
https://developer.nvidia.com/blog/grcuda-a-polyglot-language-binding-for-cuda-in-graalvm/

# GPU Execution as a DAG



We represent multi-kernel GPU computations as vertices of a DAG

- Connect kernels with data-dependencies
- Maximize parallelism, minimize synchronizations




## Use cases for multi-kernel GPU applications:

1. **GPU Graph/Database querying**
   - Union of subqueries
2. **Image processing pipelines**
   - Combine multiple filters
3. **Ensemble of ML models**
   - Combine predictions from different models on the same data

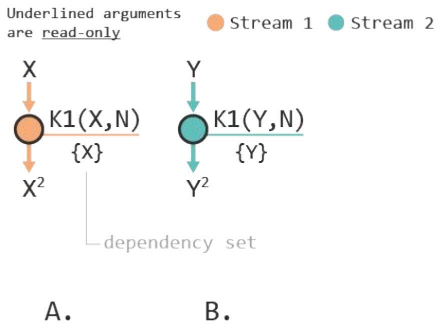
# The GrCUDA DAG computation model



## An example using the Python GrCUDA API

- All kernel invocations are **asynchronous**

```
Declare   K1 = build_kernel(K1_CODE, "square", "pointer, sint32")
kernels   K2 = build_kernel(K2_CODE, "sum",
                   "const pointer, const pointer,
                    pointer, sint32")
          X = polyglot.eval(language="grcuda", string=f"float[{N}]")
Declare   Y = polyglot.eval(language="grcuda", string=f"float[{N}]")
 arrays   Z = polyglot.eval(language="grcuda", string=f"float[1]")
          [init arrays...]
      A. — K1(NUM_BLOCKS, NUM_THREADS)(X, N)    scalar value passed by copy,
      B. — K1(NUM_BLOCKS, NUM_THREADS)(Y, N)    ignored for dependencies
```



# The GrCUDA DAG computation model



An example using the Python GrCUDA API

- All kernel invocations are **asynchronous**
- Dependencies are inferred once the computation is scheduled

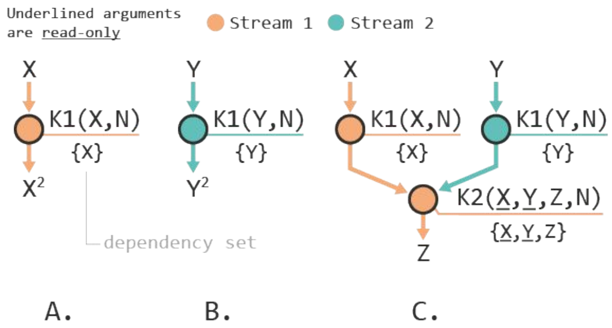
Declare kernels

```
K1 = build_kernel(K1_CODE, "square", "pointer, sint32")
K2 = build_kernel(K2_CODE, "sum",
                  "const pointer, const pointer,
                   pointer, sint32")
```

Declare arrays

```
X = polyglot.eval(language="grcuda", string=f"float[{N}]")
Y = polyglot.eval(language="grcuda", string=f"float[{N}]")
Z = polyglot.eval(language="grcuda", string=f"float[1]")
[init arrays...]
A. — K1(NUM_BLOCKS, NUM_THREADS)(X, N)
B. — K1(NUM_BLOCKS, NUM_THREADS)(Y, N)
C. — K2(NUM_BLOCKS, NUM_THREADS)(X, Y, Z, N)
```

scalar value passed by copy, ignored for dependencies



# The GrCUDA DAG computation model



An example using the Python GrCUDA API

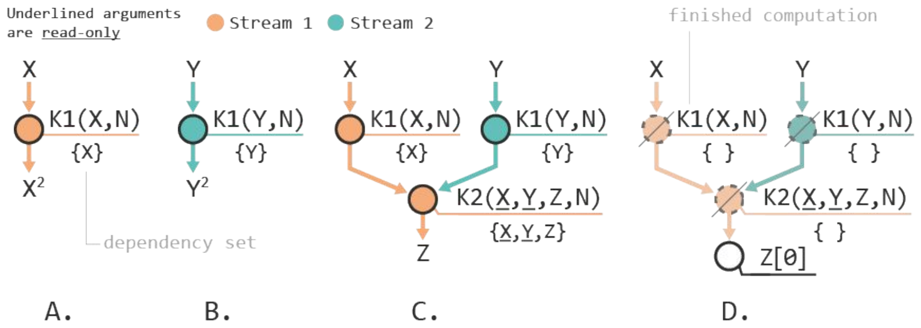
- All kernel invocations are **asynchronous**
- Dependencies are inferred once the computation is scheduled
- CPU is blocked only when it asks for results

**No user-defined dependencies in the scheduling**

```
The original API is
unmodified, everything
is transparent!
```

```
Declare   K1 = build_kernel(K1_CODE, "square", "pointer, sint32")
kernels   K2 = build_kernel(K2_CODE, "sum",
                  "const pointer, const pointer,
                   pointer, sint32")

Declare   X = polyglot.eval(language="grcuda", string=f"float[{N}]")
 arrays   Y = polyglot.eval(language="grcuda", string=f"float[{N}]")
          Z = polyglot.eval(language="grcuda", string=f"float[1]")
          [init arrays...]
     A. — K1(NUM_BLOCKS, NUM_THREADS)(X, N)    scalar value passed by copy,
     B. — K1(NUM_BLOCKS, NUM_THREADS)(Y, N)    ignored for dependencies
     C. — K2(NUM_BLOCKS, NUM_THREADS)(X, Y, Z, N)
     D. — res = Z[0] ——— synchronous CPU
                         computation
```





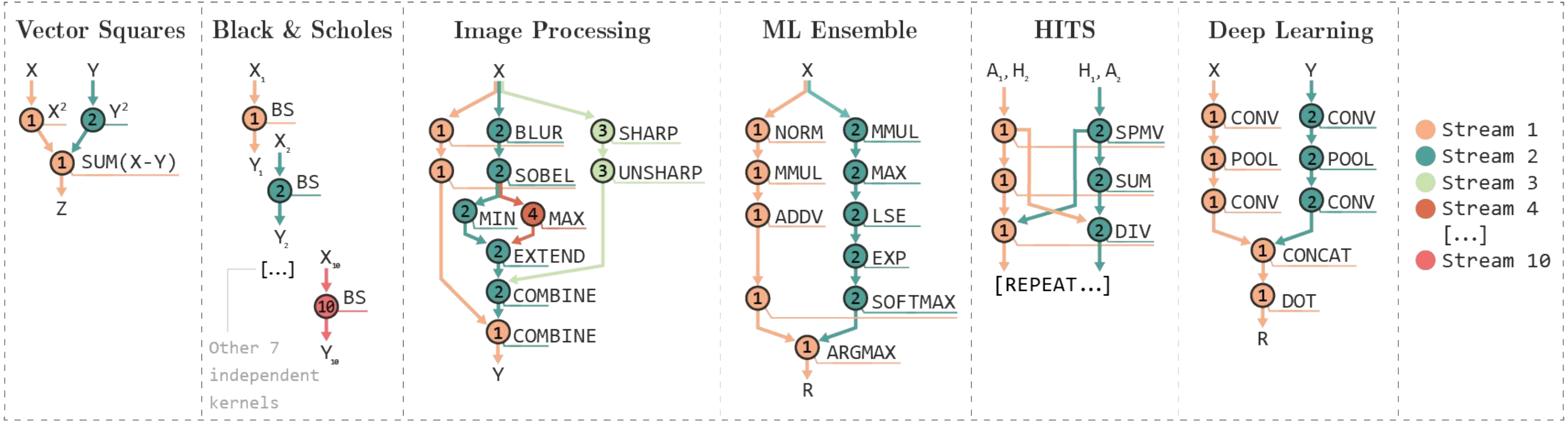

POLITECNICO MILANO 1863 NECST laboratory
POLITECNICO MILANO 1863

# Performance evaluation - Setup



6 custom benchmarks, evaluate multi-task GPU applications from different domains

- GPUs: Nvidia Tesla P100 (data-center GPU), GTX 1660 Super, GTX 960 (customer-grade GPUs)
- Note: dependency DAGs shown for clarity, but we never demand the full DAGs!



# 44% faster than synchronous execution

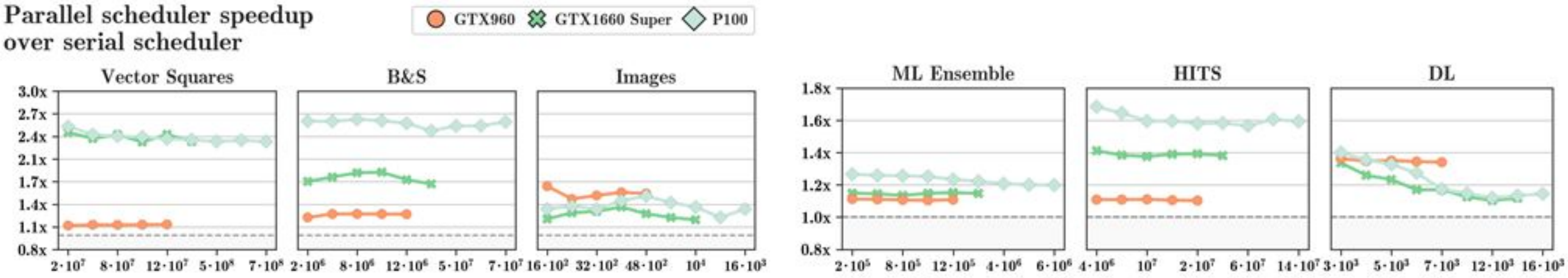


We compare against the original serial/synchronous GrCUDA scheduler

- **Always faster:** on average, **44% faster** execution time

**Same performance of CUDA Graphs** and hand-optimized CUDA events (C++ API)

- **We offer simpler scheduling at no performance loss**



# Future directions



**Started development for multi-GPU support**

- Much more complex: we need to compute data location and migration costs at run time to identify the optimal scheduling.

**Other directions**

- Applications on top of GrCUDA: e.g. sparse linear algebra,
  GrCUDA transparently maintains multiple data layouts (CSC, CSR, etc.)
- Integration with DSL: take full advantage of asynchronous execution, simplify GPU code

`Fully Open Source: github.com/AlbertoParravicini/grcuda`

- A new scheduler for GrCUDA for transparent async execution
- 44% faster than synchronous execution
- Fully integrated with GraalVM, available for Python, R, Java, JavaScript, etc.

**Fully Open Source: github.com/AlbertoParravicini/grcuda**

- We thank Oracle Labs for its support to Politecnico di Milano and its contributions to this work

Alberto Parravicini, alberto.parravicini@polimi.it
Arnaud Delamare
Marco Arnaboldi
Marco D. Santambrogio

IPDPS 2021 - 2021/05/18

POLITECNICO MILANO 1863 NECST laboratory

POLITECNICO MILANO 1863

# Enter GrCUDA, the polyglot CUDA API



**GraalVM** is a JVM that allows running Java, R, Python, JavaScript, etc. on a common backend

**GrCUDA** is a GraalVM-based DSL that exposes the CUDA API to all the languages in GraalVM

- GPU acceleration for high-level languages through a unified backend

GrCUDA is a great starting point for us

- Runtime management of arrays/kernels
- Common backend for all languages

Also many other benefits

- Simplify data-transfer with Unified Memory
- Just-In-Time CUDA compilation
- Support for any CUDA kernel and library

```
Declare  | K1 = build_kernel(K1_CODE, "square", "ptr, sint32")
kernels  | K2 = build_kernel(K2_CODE, "sum",
                    "const ptr, const ptr, ptr, sint32")
         | X = polyglot.eval("grcuda", "float[{}]".format(N))
Declare  | Y = polyglot.eval("grcuda", "float[{}]".format(N))
 arrays  | Z = polyglot.eval("grcuda", "float[1]")
           [init arrays...]
      A. — K1(NUM_BLOCKS, NUM_THREADS)(X, N)
      B. — K1(NUM_BLOCKS, NUM_THREADS)(Y, N)
      C. — K2(NUM_BLOCKS, NUM_THREADS)(X, Y, Z, N)
      D. — res = Z[0]
```

# Differences with Existing Techniques



Many libraries provide APIs for GPU scheduling: TensorFlow, CUDA Graphs, and more [1,2]

**What's new here?**

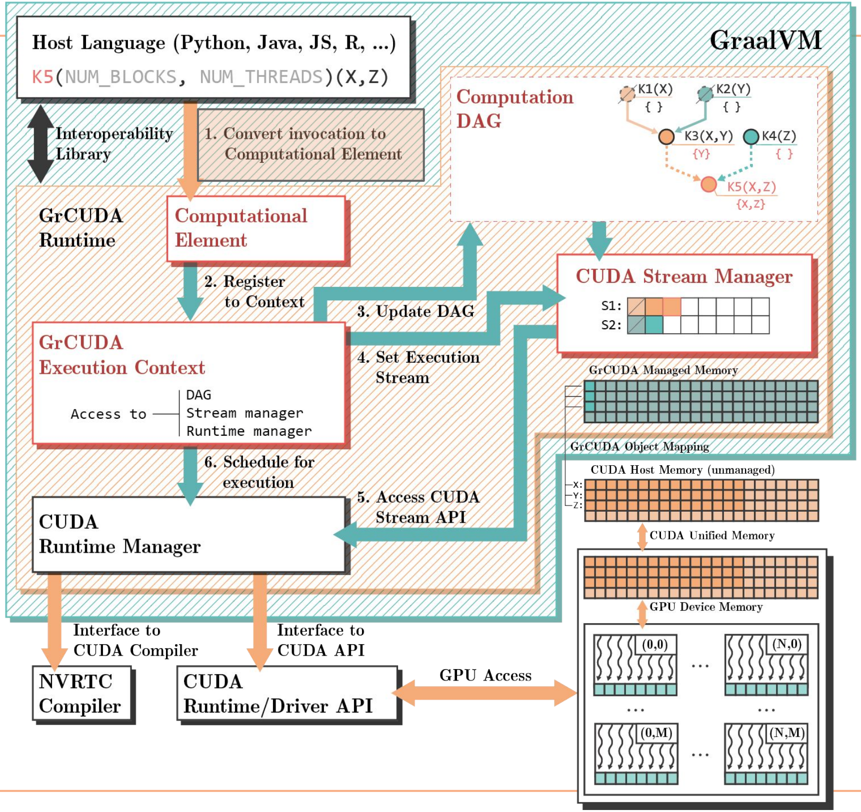
1. It's **fully transparent** to the user, the API of GrCUDA is not modified
2. Dependencies are **computed at runtime**, not at compile time or eagerly
   - GraalVM partial evaluation minimizes the runtime overheads (e.g. repeated array accesses)
3. Updates to the GrCUDA runtime are immediately available to every GraalVM language
   - Instead of having different libraries: *PyCUDA, JCuda, GPU.js, etc.*

[1] Gautier, Thierry, et al. "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures." *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013.
[2] Fumero, Juan, et al. "Dynamic application reconfiguration on heterogeneous hardware." *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2019.

# The extended GrCUDA architecture



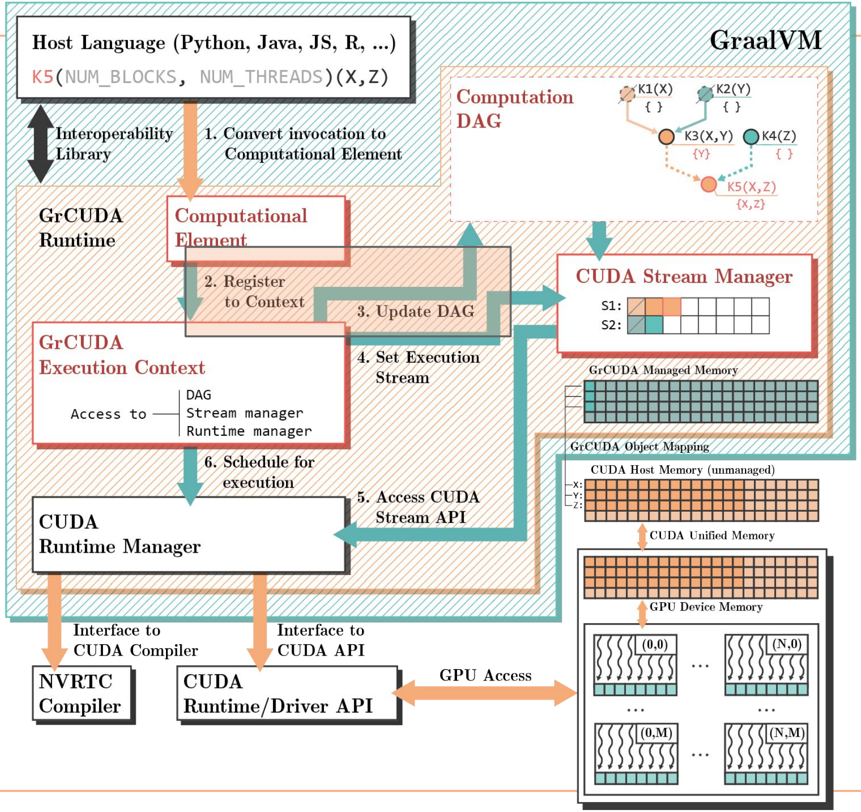
- New components are highlighted in red
- Kernel invocations are wrapped into *computational elements* **(1)**



# The extended GrCUDA architecture



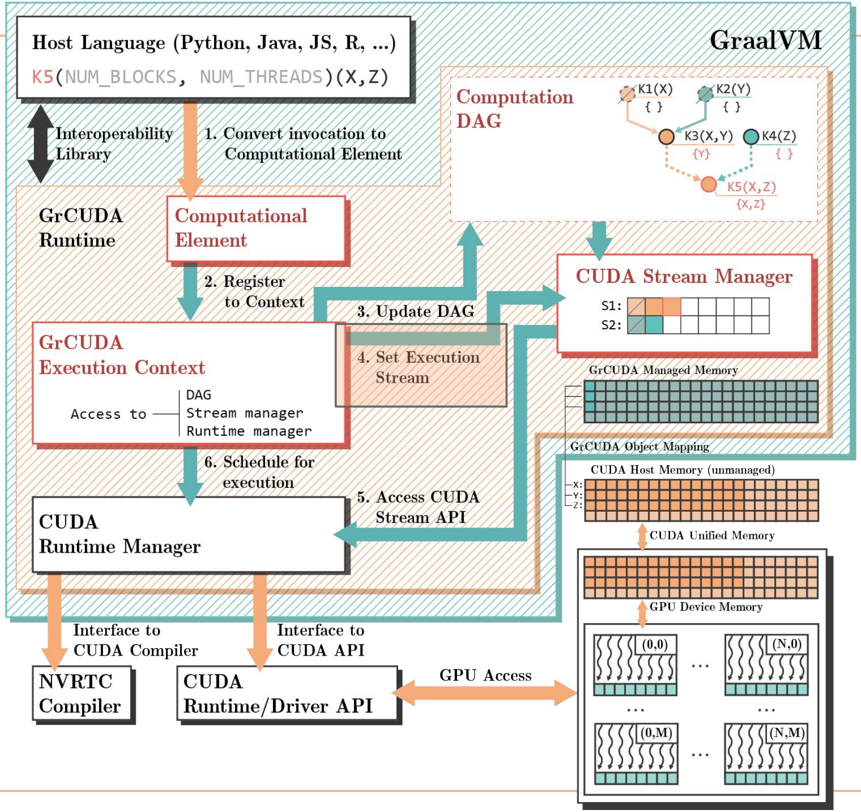
- New components are highlighted in red
- Kernel invocations are wrapped into *computational elements* **(1)**
- The GrCUDA *execution context* computes data-dependencies, updates the DAG **(2, 3)**



# The extended GrCUDA architecture



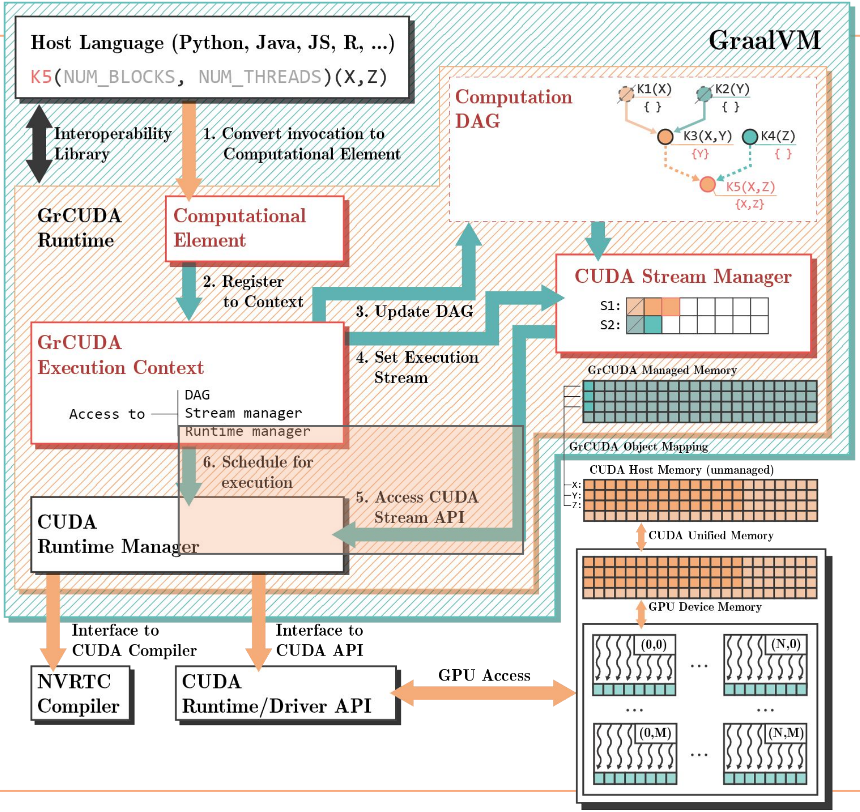
- New components are highlighted in red
- Kernel invocations are wrapped into *computational elements* **(1)**
- The GrCUDA *execution context* computes data-dependencies, updates the DAG **(2, 3)**
- The computation is assigned a *CUDA stream* based on dependencies and availability **(4)**



# The extended GrCUDA architecture



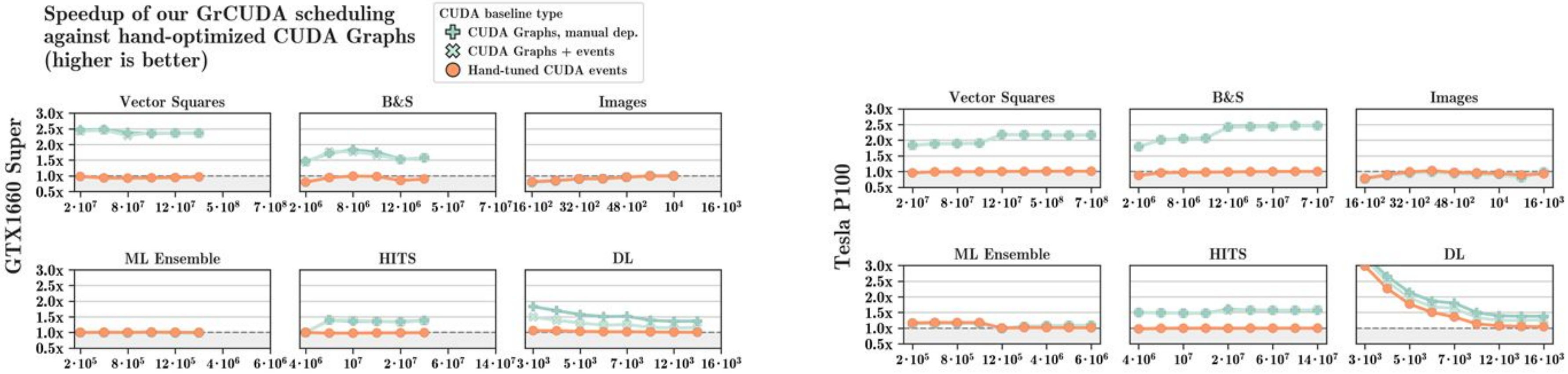
- New components are highlighted in red
- Kernel invocations are wrapped into *computational elements* **(1)**
- The GrCUDA *execution context* computes data-dependencies, updates the DAG **(2, 3)**
- The computation is assigned a *CUDA stream* based on dependencies and availability **(4)**
- The *execution context* schedules the computation on GPU **(5, 6)**
  - Data prefetching and event synchronizations are non-blocking and asynchronous



# Performance against CUDA Graphs



- We are not slower (and often faster) than the highly optimized CUDA Graphs, which requires manual dependencies. We have also the same performance as hand-optimized scheduling with CUDA events
- **We offer simpler scheduling at no performance loss**



# Unlocking better GPU utilization

Our scheduler exploits untapped GPU resources

- Higher values for memory throughput, L2 cache utilization, etc.
- Significant overlap between transfer and computations