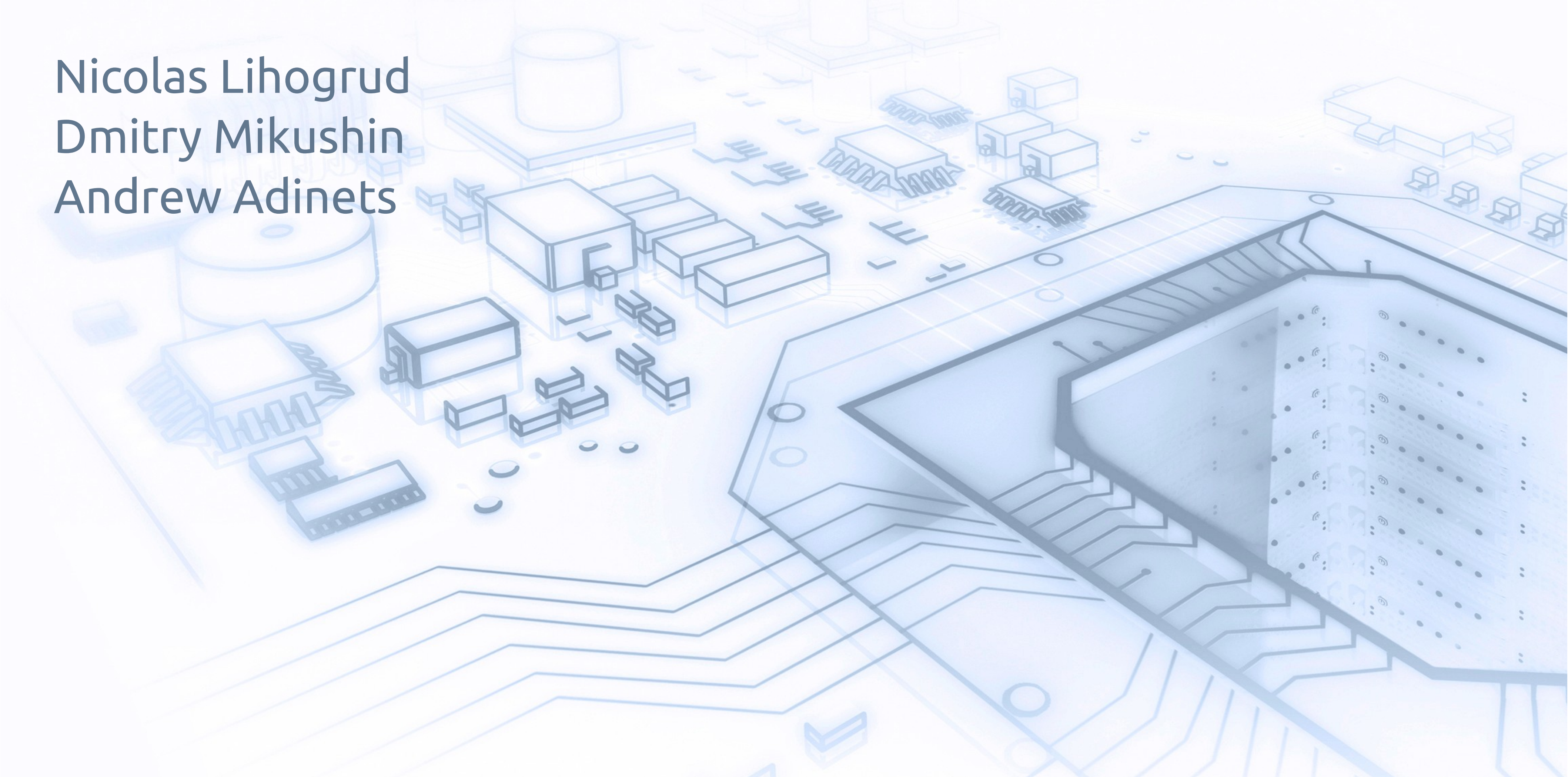
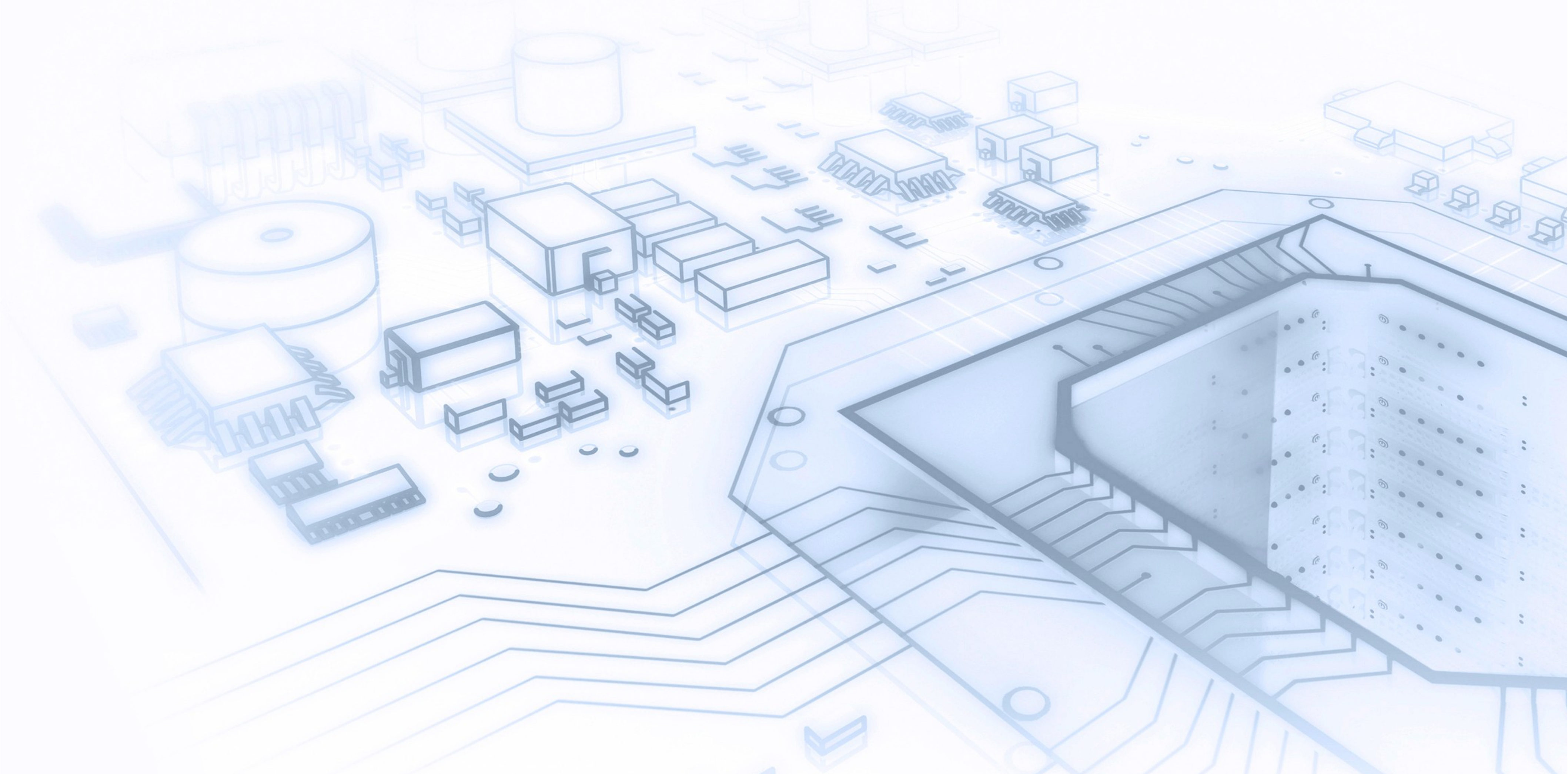
PARALLEL COMPUTATIONAL TECHNOLOGIES '2012

KernelGen – a toolchain for automatic GPU-centric applications porting

Nicolas Lihogrud
Dmitry Mikushin
Andrew Adinets



Contents

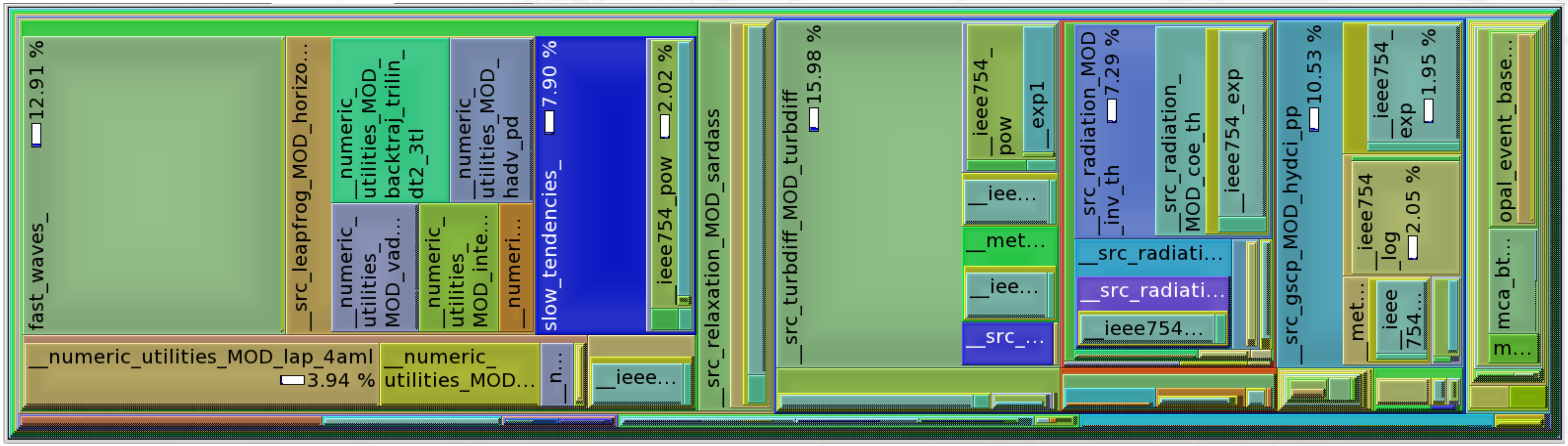


Motivation and targets

Motivation

- The need to port huge numerical models on GPU: no computational "kernel", individual model blocks have too small self perf impact (~**10%**), resulting into small speedups, if only one block is ported (the figure below is for COSMO model)

  → too hard to port models on GPU by hand



Motivation

- Most of the modern numerical weather prediction models are not suitable for manual parallelization due to enormous code base size:

  → too hard to port models on GPU by hand

Model	Developer	Lines of code
COSMO 4.13	DWD	187K
WRF 3.3	NOAA/NCAR	370K
AROME/ARPEGE	METEOFRANCE	2280K

Motivation

- In well-known study on porting WSM5 block of the WRF model with OpenACC directives 40-60% of time is spent on communications

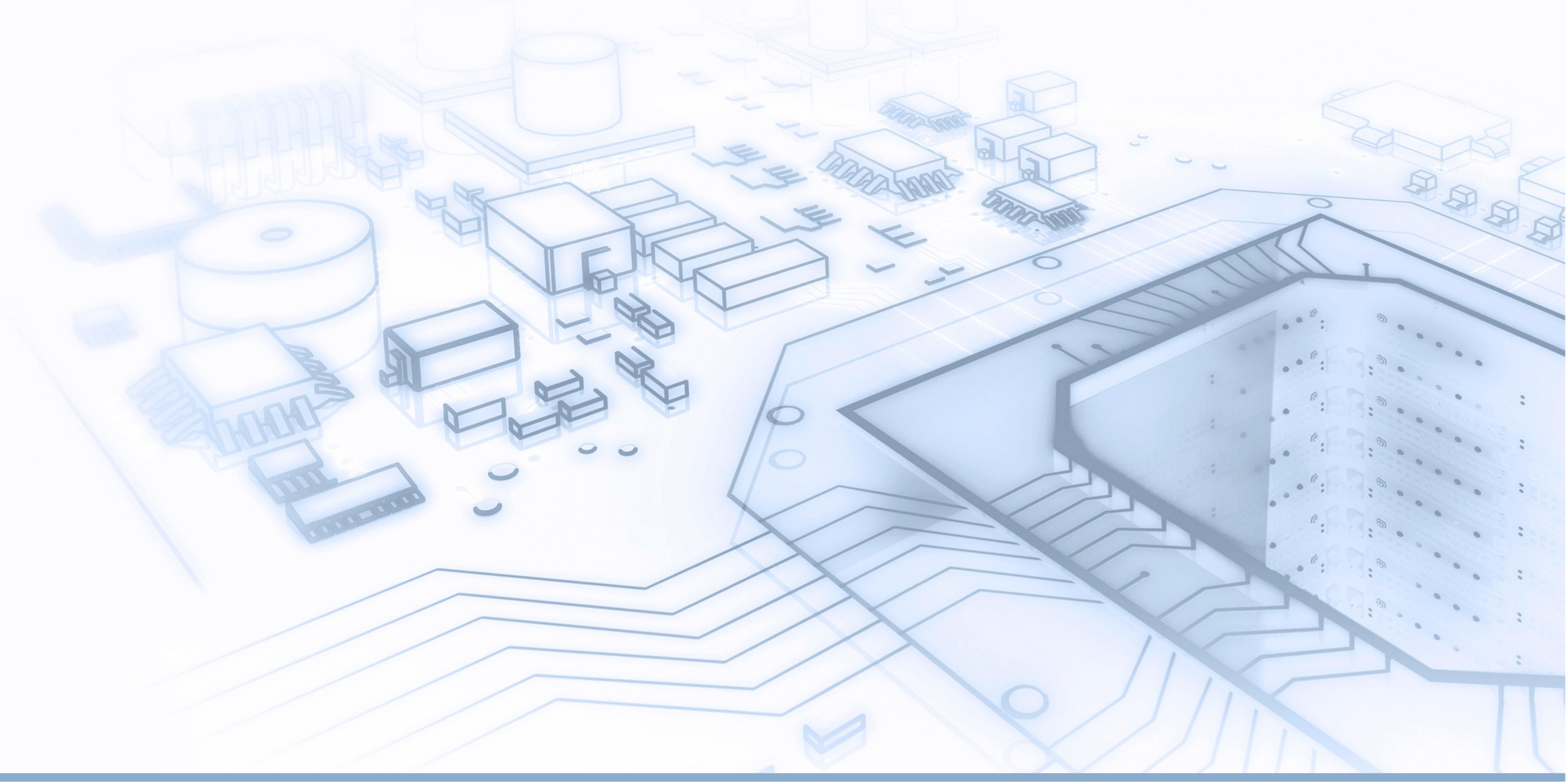
  → intensive CPU ⟷ GPU data transfer may introduce significantly negative performance impact

The number of CPU cores and GPUs	Total time (seconds)	The time of transfers	The time of computations
1 / -	236		
4 / -	70		
1 / 1	19.72	10.75	8.85
2 / 2	12	6.87	5.29

Michael Wolfe, Craig Toepfer – Porting WRF

Targets

- Create runtime environment capable of executing original CPU applications on the GPU **without code modifications**
- Minimize data transfers by keeping all the application data in GPU memory and automatically porting onto GPU as much code as possible



Idea

Idea

- Many computational kernels, single main kernel
- Minimize data transfers by performing some serial computations on the GPU

\+ Use DragonEgg/LLVM/Polly:

- ➔ Support many programming languages
- ➔ Automated search for potentially parallel code blocks
- ➔ Automated generation of GPU kernels

Idea

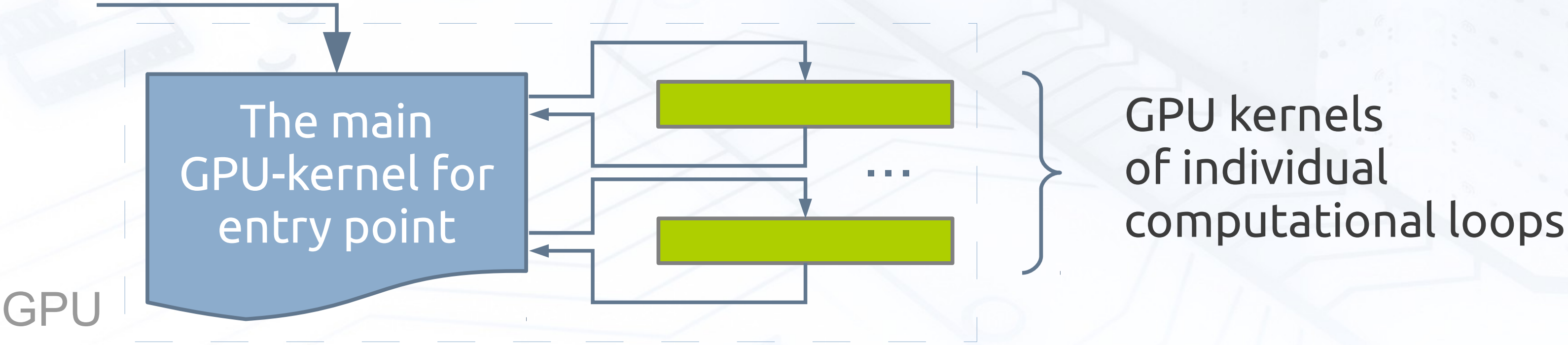
- Main host system and peripheral GPU:
  - ➔ Transfer data from host to GPU
  - ➔ Launch the GPU kernel and wait for completion
  - ➔ Transfer results back from GPU to host

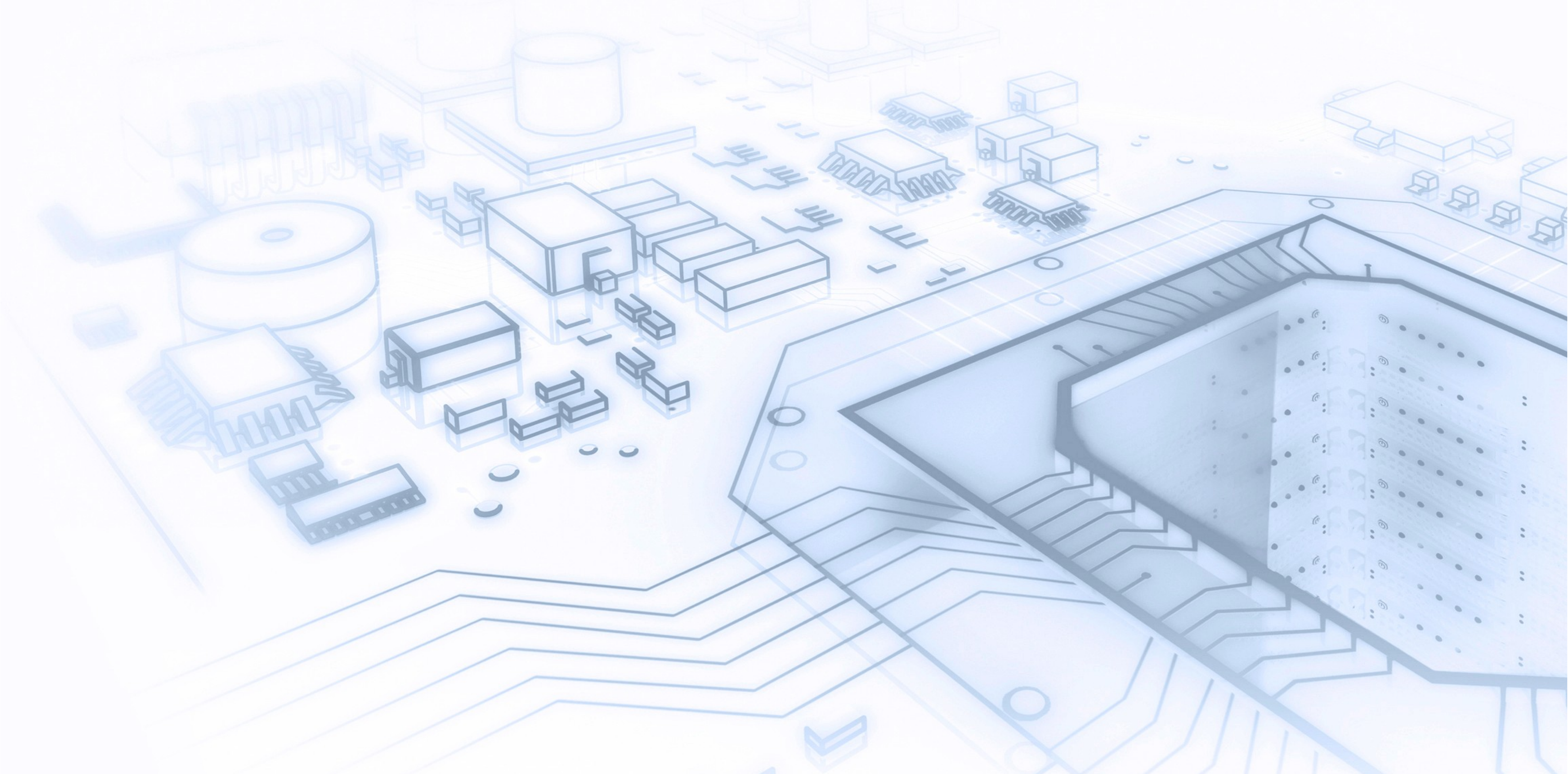
Idea

- ~~Main host system and peripheral GPU:~~
  - ➔ ~~Transfer data from host to GPU~~
  - ➔ ~~Launch the GPU kernel and wait for completion~~
  - ➔ ~~Transfer results back from GPU to host~~

Idea

- ~~Main host system and peripheral GPU:~~
  - ~~➔ Transfer data from host to GPU~~
  - ~~➔ Launch the GPU kernel and wait for completion~~
  - ~~➔ Transfer results back from GPU to host~~
- Main GPU and peripheral host-system:
  - ➔ Port on GPU as much source code as possible, except host-only functions (I/O, syscalls, …)





Key results

Results

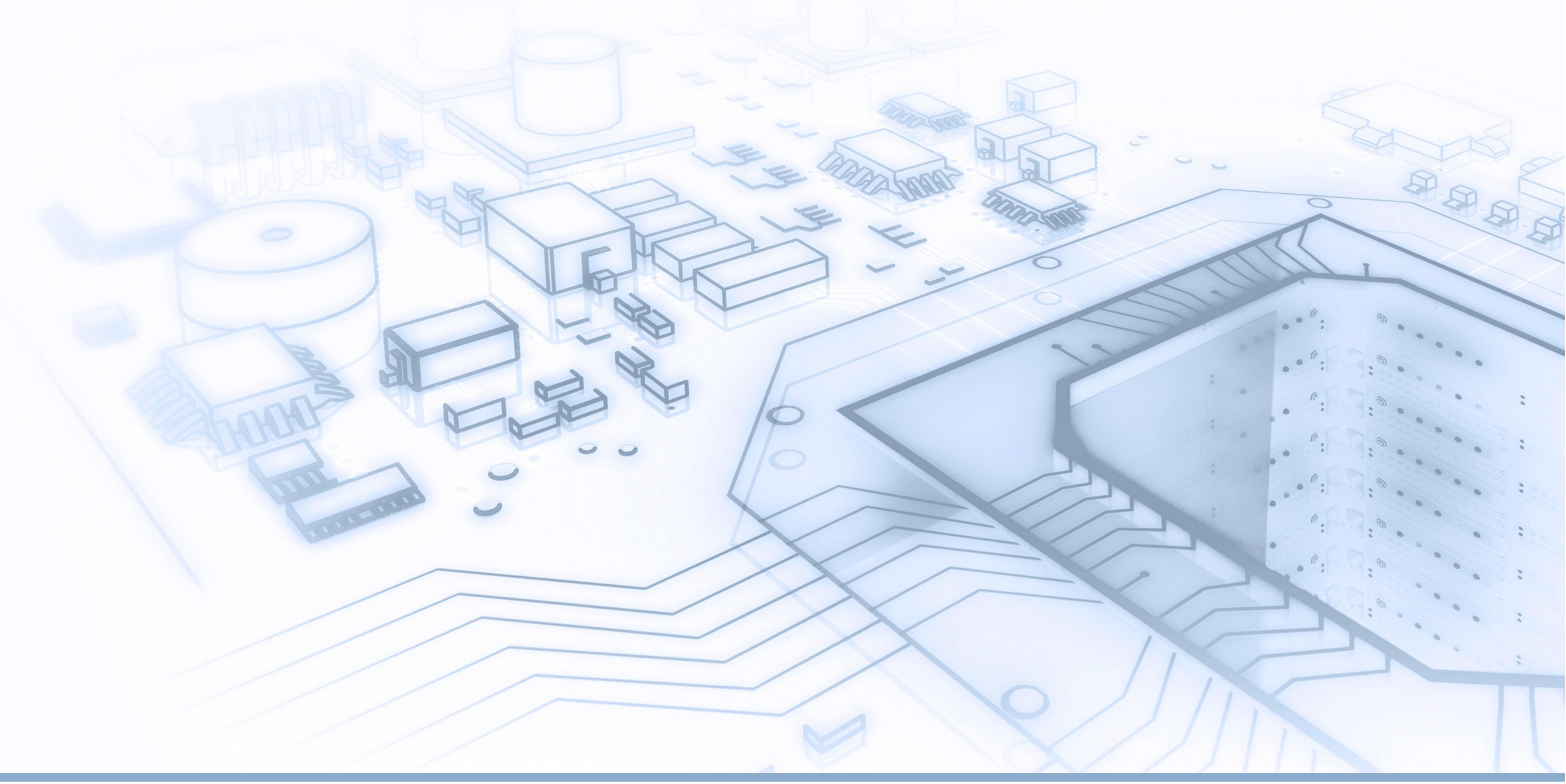
- KernelGen can recognize parallel loops in simple tests and can generate GPU kernels with efficiency comparable to PGI Accelerator
- KernelGen successfully compiles major NWP models: WRF, COSMO and SLM (ПЛАВ)
- Complexity: half-year work of 2 developers, based on many other projects

Some tests

Performance and register footprint of CUDA kernels generated with KernelGen and PGI:

Kernel	KGen time	KGen #regs	PGI time	PGI #regs	CPU time
sincos	0.00390	9	0.00357	29	0.64972
jacobi_1	0.02063	24	0.02862	17	1.05245
jacobi_2	0.01233	7	0.01206	10	0.10387

In some cases kernels generated by KernelGen and PGI are significantly different, but performance is always comparable



KernelGen

from the user point of view

User interface

- KernelGen can generate GPU code out of any language supported by the gcc compiler frontend
- KernelGen compiler flags are fully compatible with gcc (except the LTO logic, which is replaced by custom IR-code link step)
- Required additional GPU configuration parameters are provided through the environment variables (kernelgen_runmode, kernelgen_verbose, kernelgen_szheap, ...)

Compile application for KernelGen

```
$ make
cd kernelgen && make
make[1]: Entering directory
`/home/marcusmae/Programming/kernelgen/tests/perf/sincos/m
alloc/kernelgen'
kernelgen-gfortran -c ../sincos.f90 -o sincos.o
KernelGen : NumExtractedLoops = 1
CurrentFunction:"sincos_" CurrentHeader:"21.orig.header"
KernelGen : NumExtractedLoops = 2
CurrentFunction:"sincos_" CurrentHeader:"12.orig.header"
KernelGen : NumExtractedLoops = 3
CurrentFunction:"sincos_" CurrentHeader:"3.orig.header"
kernelgen-gcc -std=c99 -c ../main.c -o main.o
KernelGen : NumExtractedLoops = 1 CurrentFunction:"main"
CurrentHeader:"6.orig.header"
kernelgen-gfortran sincos.o main.o -o sincos
$
```

Launch application for KernelGen

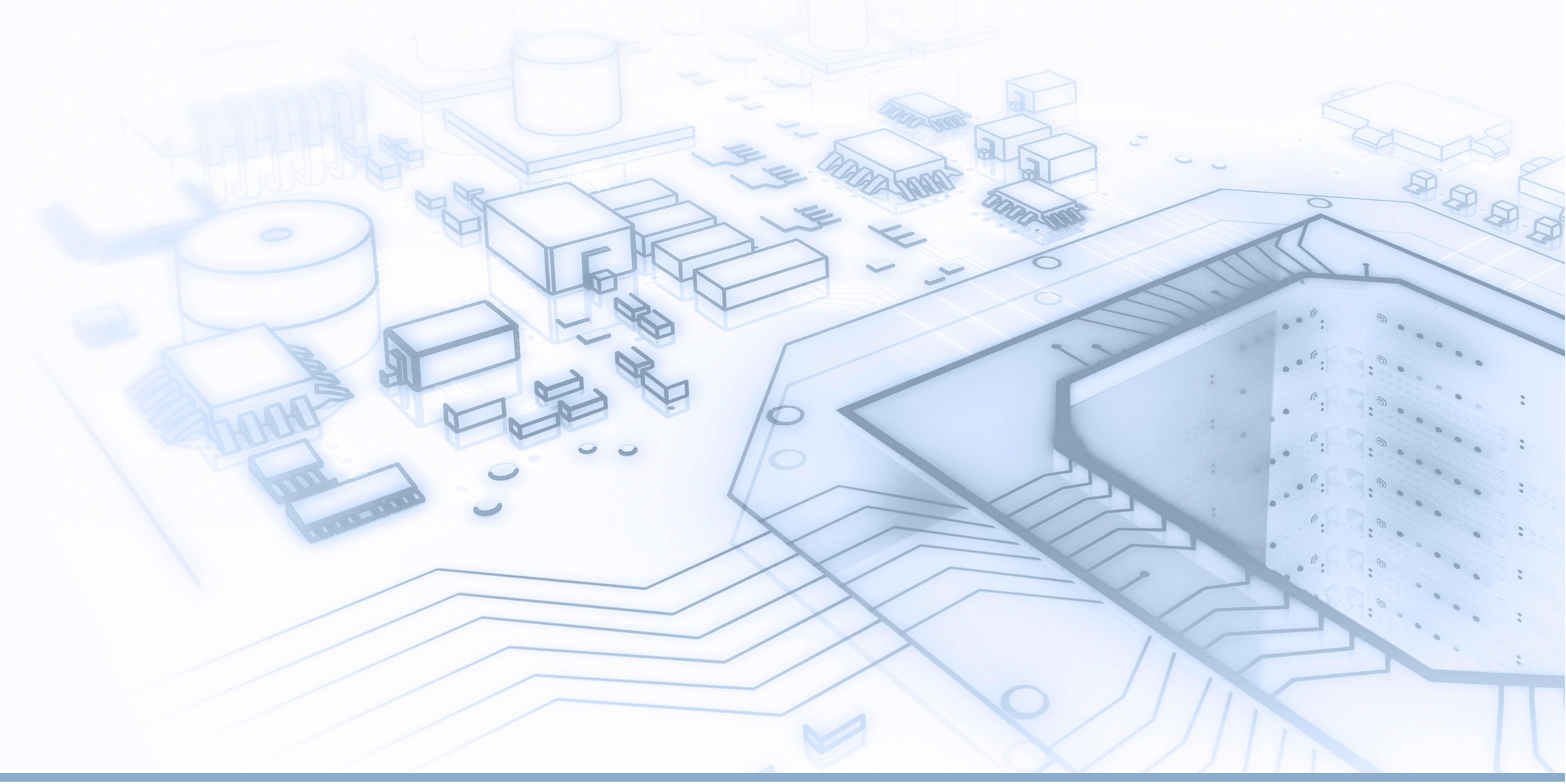
```
[marcusmae@noisy malloc]$ kernelgen_verbose=1
kernelgen_runmode=1 kernelgen_szheap=$((1024*1024*1024))
kernelgen/sincos 512 512 64
Using KernelGen/CUDA

Building kernels index ...
__kernelgen_sincos__loop_21
__kernelgen_sincos__loop_12
__kernelgen_sincos__loop_3
__kernelgen_main_loop_6
__kernelgen_main

...

Launching kernel __kernelgen_sincos__loop_3
    blockDim = { 32, 16, 2 }
    gridDim = { 16, 32, 32 }
__kernelgen_sincos__loop_3 time = 0.00407996 sec
only the kernel execution time = 0.00390173 sec
```

Compile application for PGI

```
$ make
cd pgi && make
make[1]: Entering directory
`/home/marcusmae/Programming/kernelgen/tests/perf/sincos/malloc/pgi'
pgfortran -fast -Mnomain -Minfo=accel -ta=nvidia:4.1,time
-Mcuda=keepgpu,keepbin,keepptx -c ../sincos.f90 -o sincos.o
sincos:
     12, Generating copyin(y(:nx,:ny,:nz))
         Generating copyin(x(:nx,:ny,:nz))
         Generating copyout(xy1(:nx,:ny,:nz))
     13, Loop is parallelizable
     14, Loop is parallelizable
     15, Loop is parallelizable
         Accelerator kernel generated
         13, !$acc do parallel, vector(4) ! blockidx%y threadidx%z
         14, !$acc do parallel, vector(4) ! blockidx%x threadidx%y
         15, !$acc do vector(16) ! threadidx%x
pgcc -c ../main.c -o main.o
pgfortran -fast -Mnomain -Minfo=accel -ta=nvidia:4.1,time
-Mcuda=keepgpu,keepbin,keepptx sincos.o main.o -o sincos
make[1]: Leaving directory
`/home/marcusmae/Programming/kernelgen/tests/perf/sincos/malloc/pgi'
$
```

Launch application for PGI

```
$ pgi/sincos 512 512 64
Accelerator Kernel Timing data
home/marcusmae/Programming/kernelgen/tests/perf/sincos/mal
loc/pgi/../sincos.f90
  sincos
    12: region entered 1 time
        time(us): total=2362577 init=2268435 region=94142
                  kernels=3575 data=86839
        w/o init: total=94142 max=94142 min=94142
avg=94142
        15: kernel launched 1 times
            grid: [128x16]  block: [16x4x4]
            time(us): total=3575 max=3575 min=3575
avg=3575
```



KernelGen

from the developer point of view

Compiling and linking

- Port whole code on the GPU
  - ➔ Compiling and linking
  - ➔ Problem: no linker for GPU code
    - ➔ Compile parts of the code for GPU and link them manually, inlining everything into main kernel and loops kernels
    - ➔ CUDA-compiler exists only for C/C++ code
- Automatically extract parallel loops
  - ➔ Analyze simple AST or specialized IR?
    - ➔ Parse high-level language or its AST (Rose, XML) → waste of time
    - ➔ Parse intermediate representation (LLVM IR)

Components

- **LLVM** – the modular analysis and transformation system, working with the specialized intermediate representation (LLVM IR)
- **DragonEgg** – the gcc plugin, which converts gimple into LLVM IR, i.e. original program may be written in any language supported by gcc
- **Polly** – the loop optimization engine for LLVM IR
- **C Backend** – generates C code for the given LLVM IR module

Application runtime design

- Binary image still contains fully working host code used by default; GPU version is activated by request
- Almost all the code is running on the GPU
  - ➔ Some parts of the code is not possible to port (functions in external libraries, system calls) → external host calls are supported
- All data is stored in GPU memory and is offloaded to host only by request
- The main kernel is persistent on the GPU during the whole application lifetime → hacks to overcome some limitations of CUDA, concurrent kernels execution

Handling host-only calls

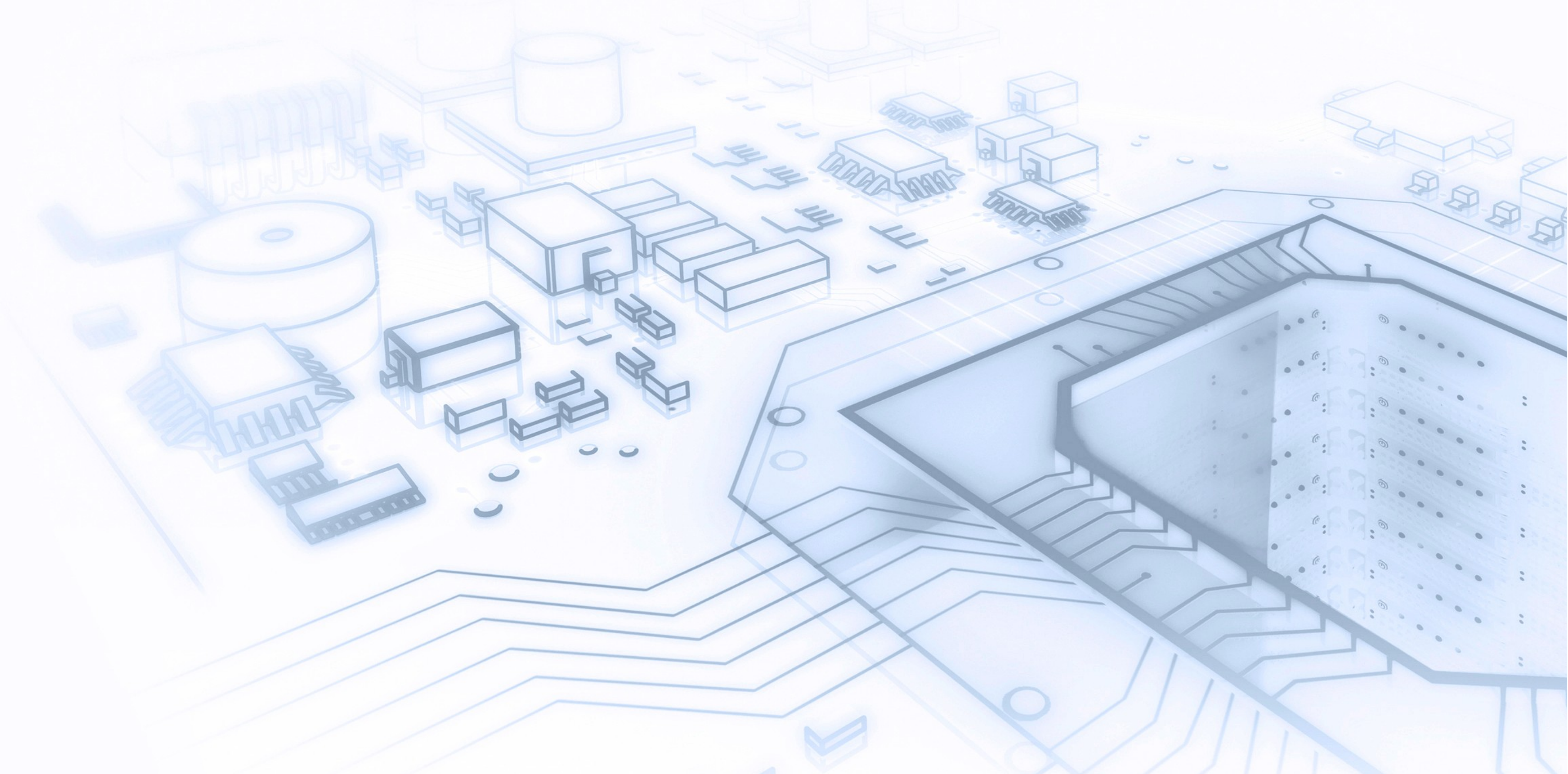
- Example: `atoi(<address_in_gpu>);`
- Launch CPU function and synchronize data by request
  - ➔ SIGSEGV handler to catch GPU memory range accesses during the host call
  - ➔ mmap host memory pages onto GPU memory ranges and fill them with GPU data; synchronize data back after the function is finished
- Interaction model: “active” GPU, “passive” host (GPU initiates kernels launches and host calls)

Generating parallel loops

- Loop iterations dependency checks
  - ➔ Runtime Alias Analysis
  - ➔ LLVM Polly, ClooG
- Generate LLVM IR for GPU kernels
  - ➔ GPU extensions for Polly
- Determine the GPU kernel compute grid
  - ➔ Substitute kernel arguments
- Runtime-optimization of loops GPU kernels
  - ➔ Substitute compute grid parameters, optimize LLVM
  - ➔ Cache the analysis results

KernelGen uses JIT-compilation

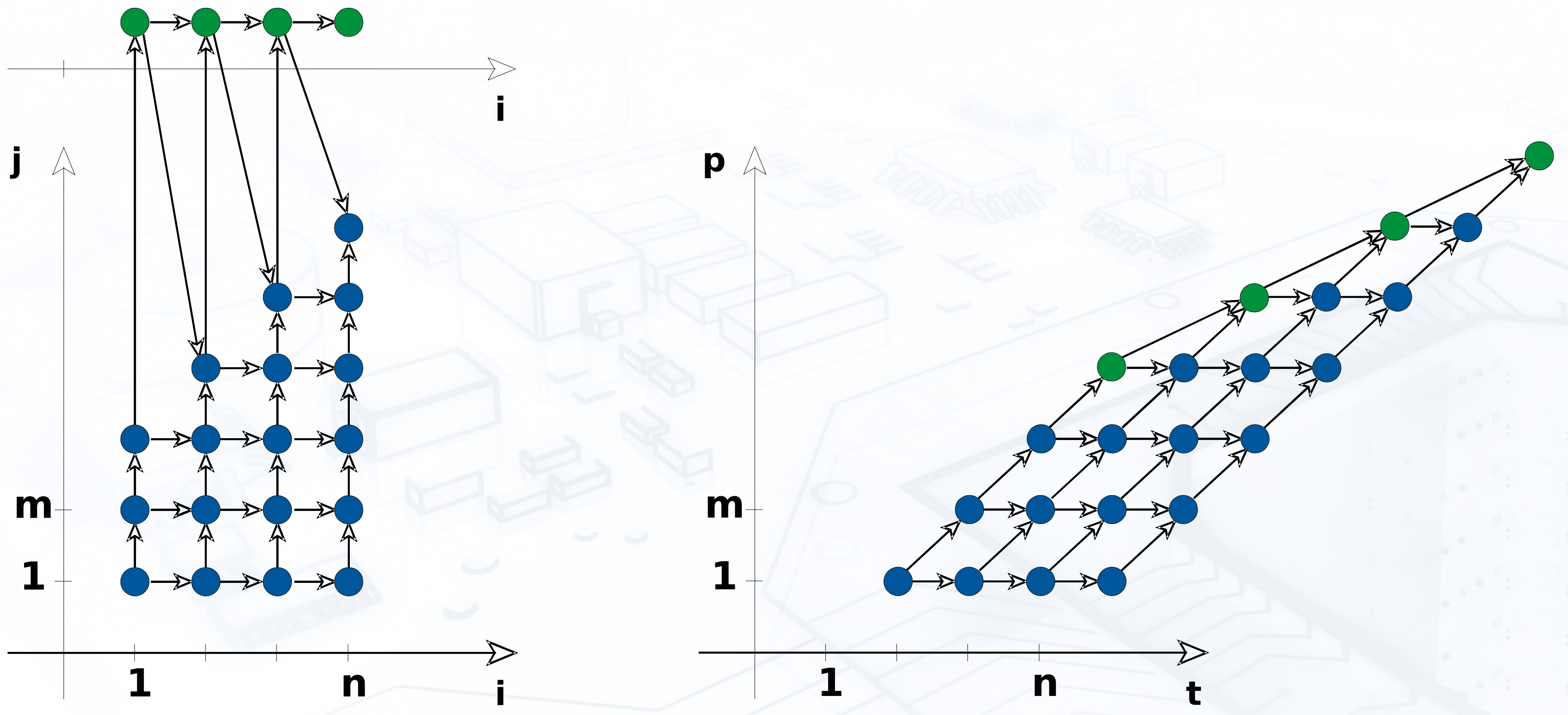
- Kernel arguments substitution
  - ➔ Runtime Alias Analysis
  - ➔ Determine the optimal compute grid parameters
- Substitute the compute grid parameters
  - ➔ Helps to reduce the register footprint



LLVM Polly

LLVM Polly: a tool for loop transformations

- Polly is able to transform loop with dependent iterations (a) into loop with parallel iterations (b):

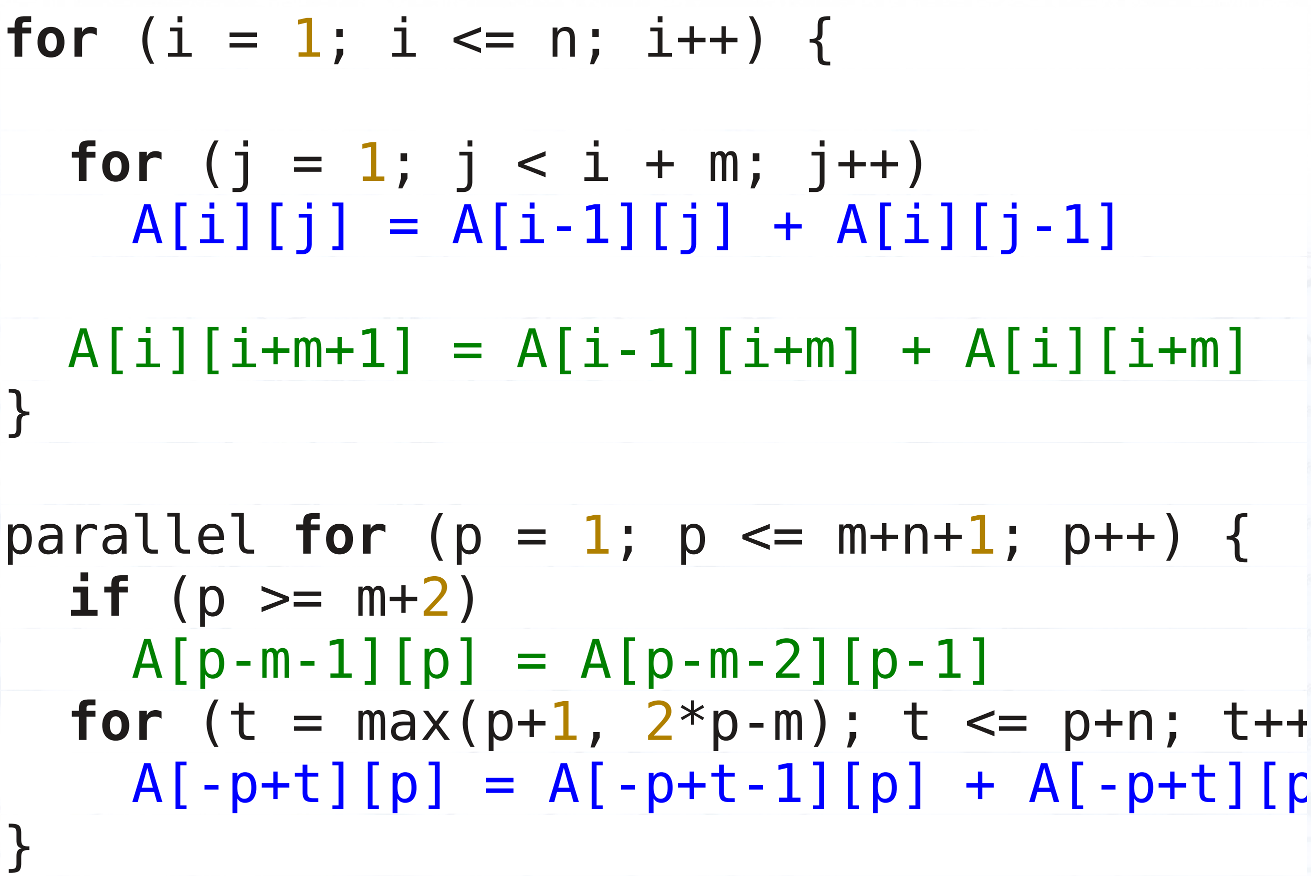


(a) (b)

Illustrations (c) Tobias Grosser

LLVM Polly: a tool for loop transformations

- Polly is able to transform loop with dependent iterations (a) into loop with parallel iterations (b):



(a)

```
for (i = 1; i <= n; i++) {

  for (j = 1; j < i + m; j++)
    A[i][j] = A[i-1][j] + A[i][j-1]

  A[i][i+m+1] = A[i-1][i+m] + A[i][i+m]
}
```

(b)

```
parallel for (p = 1; p <= m+n+1; p++) {
  if (p >= m+2)
    A[p-m-1][p] = A[p-m-2][p-1]
  for (t = max(p+1, 2*p-m); t <= p+n; t++
    A[-p+t][p] = A[-p+t-1][p] + A[-p+t][
}
```

Code examples (c) Tobias Grosser

LLVM Polly: capabilities

- Search for valid SCoPs in LLVM IR
- Optimizations:
  - ➔ Split loops
  - ➔ Transform loops polyhedra
  - ➔ Loops interchanging
  - ➔ ...
- Detect parallel loops
- Generate CLooG AST
- Generate LLVM IR out of CLooG AST

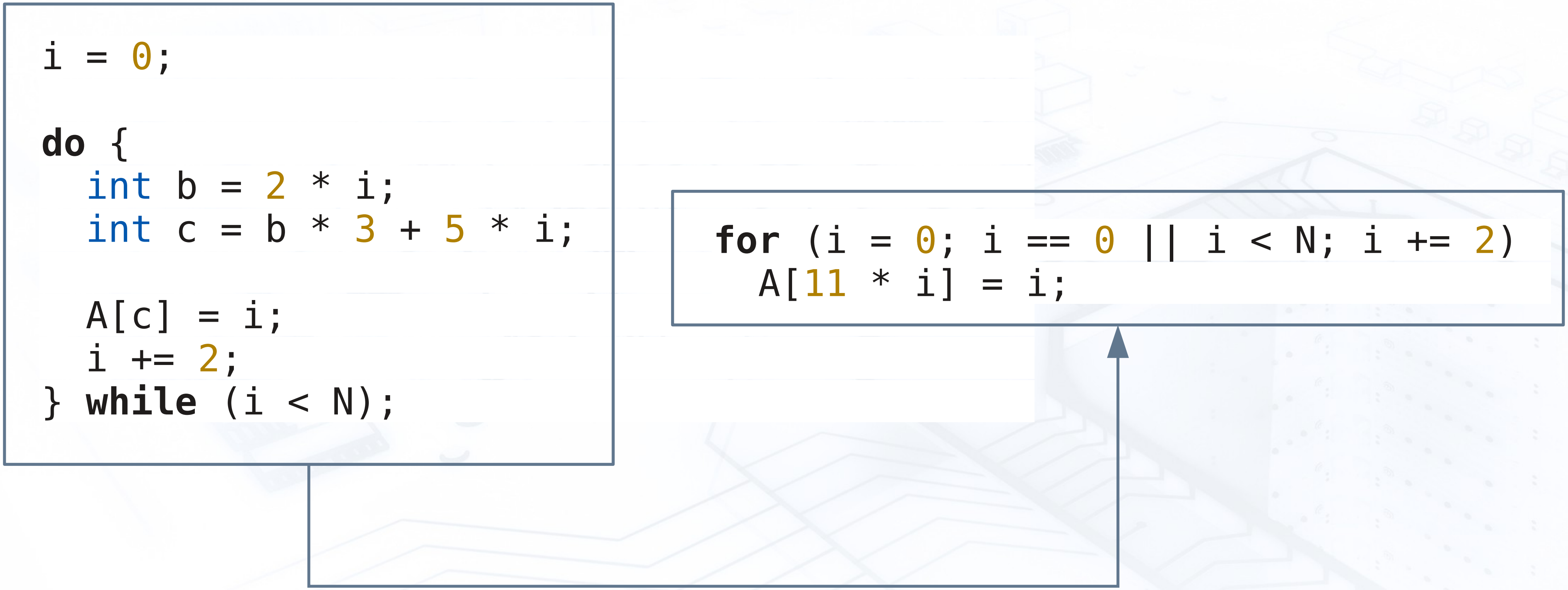
🟩 – used by KernelGen

Static Control Part (SCoP)

- Polly works with ScoPs – parts of the program with the following properties:
    - ➔ Structured control flow – counting variables, conditions
    - ➔ Loop boundaries, array indices and conditionals are affine expressions of parameters and induction variables
    - ➔ All operations (including function calls) do not have side effects
- For ScoPs it is possible to determine:
    - ➔ Iterations polyhedras and the order of iterations inside them
    - ➔ Memory access patterns

Semantic SCoP

- Polly can detect the semantic ScoPs, i.e. not only indexed for-/do- loops, but any code parts that behave like indexed for-/do- loops:



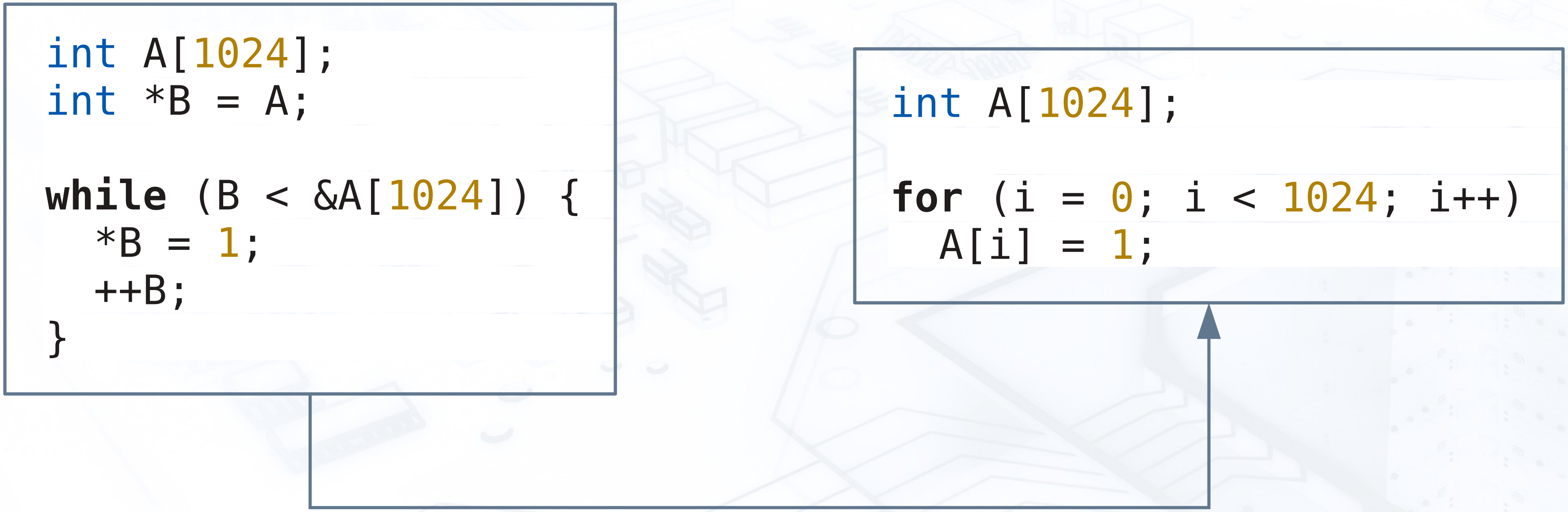
```
i = 0;

do {
  int b = 2 * i;
  int c = b * 3 + 5 * i;

  A[c] = i;
  i += 2;
} while (i < N);
```

```
for (i = 0; i == 0 || i < N; i += 2)
  A[11 * i] = i;
```

Code examples (c) Tobias Grosser

Semantic SCoP

- Polly can detect the semantic ScoPs, i.e. not only indexed for-/do- loops, but any code parts that behave like indexed for-/do- loops:



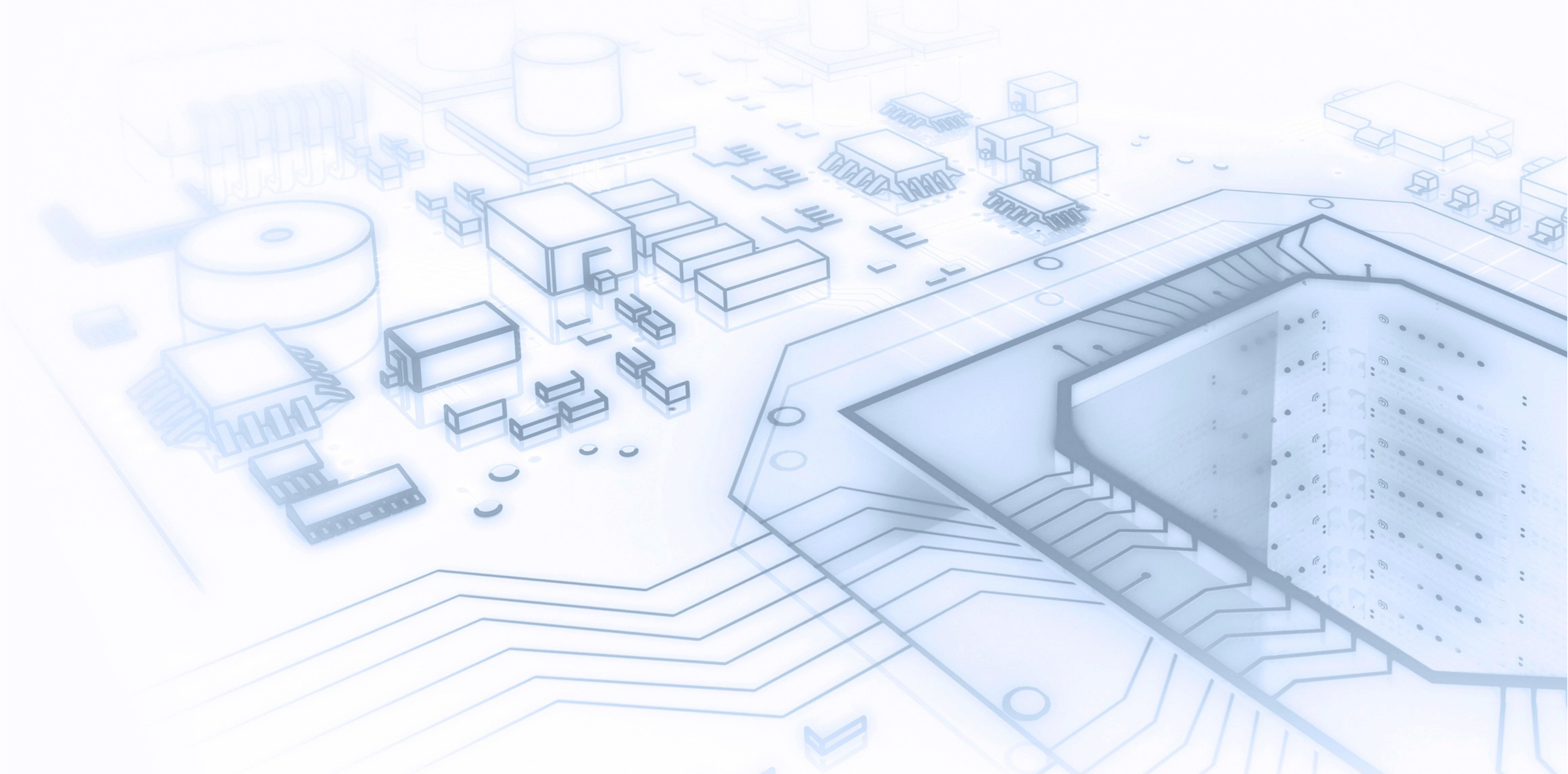
```
int A[1024];
int *B = A;

while (B < &A[1024]) {
  *B = 1;
  ++B;
}
```

```
int A[1024];

for (i = 0; i < 1024; i++)
  A[i] = 1;
```

Code examples (c) Tobias Grosser

Polly is not enough!

- In real-world applications compile-time info is often insufficient to detect the parallel loop



Runtime Alias Analysis

Runtime Alias Analysis

- In runtime more accurate alias analysis could be performed after substituting the values of pointers and checking the used memory intervals for intersection
  - ➔ An argument to postpone loop analysis for runtime (JIT-compilation)
- For each pointer operation the access function is computed:

  `f(<the_number_of_iteration>)`

- With help of the ISL library, the problem of linear programming is solved: *find the **f** maximum and minimum values in the given iterations space*

Runtime Alias Analysis

- After substitution the pointer value in known, as well as its maximum and minimum relative offsets

➔ Thus, it is possible to compose the memory intervals:

```
0 <= i < 100
0 <= j < 200
ptr = (int*)322636916
f(i,j) = i * 200 + j
```

→

```
f_minimum = 0
f_maximum = 19999
interval:
   [322636916, 322716914]
```

Runtime Alias Analysis

- For each pointer use we determine the memory interval
- If some memory write interval
  - ➔ Intersects with another write interval
  - ➔ Or intersects with read interval

Then the kernel is not parallel!

Generating CUDA kernels

- Polly codegen was modified in order to:
  - ➔ Determine the GPU thread indices in compute grid (GridDim, BlockDim, BlockIdx, ThreadIdx)
  - ➔ Map loop iterations space onto GPU threads space

Generating CUDA kernels

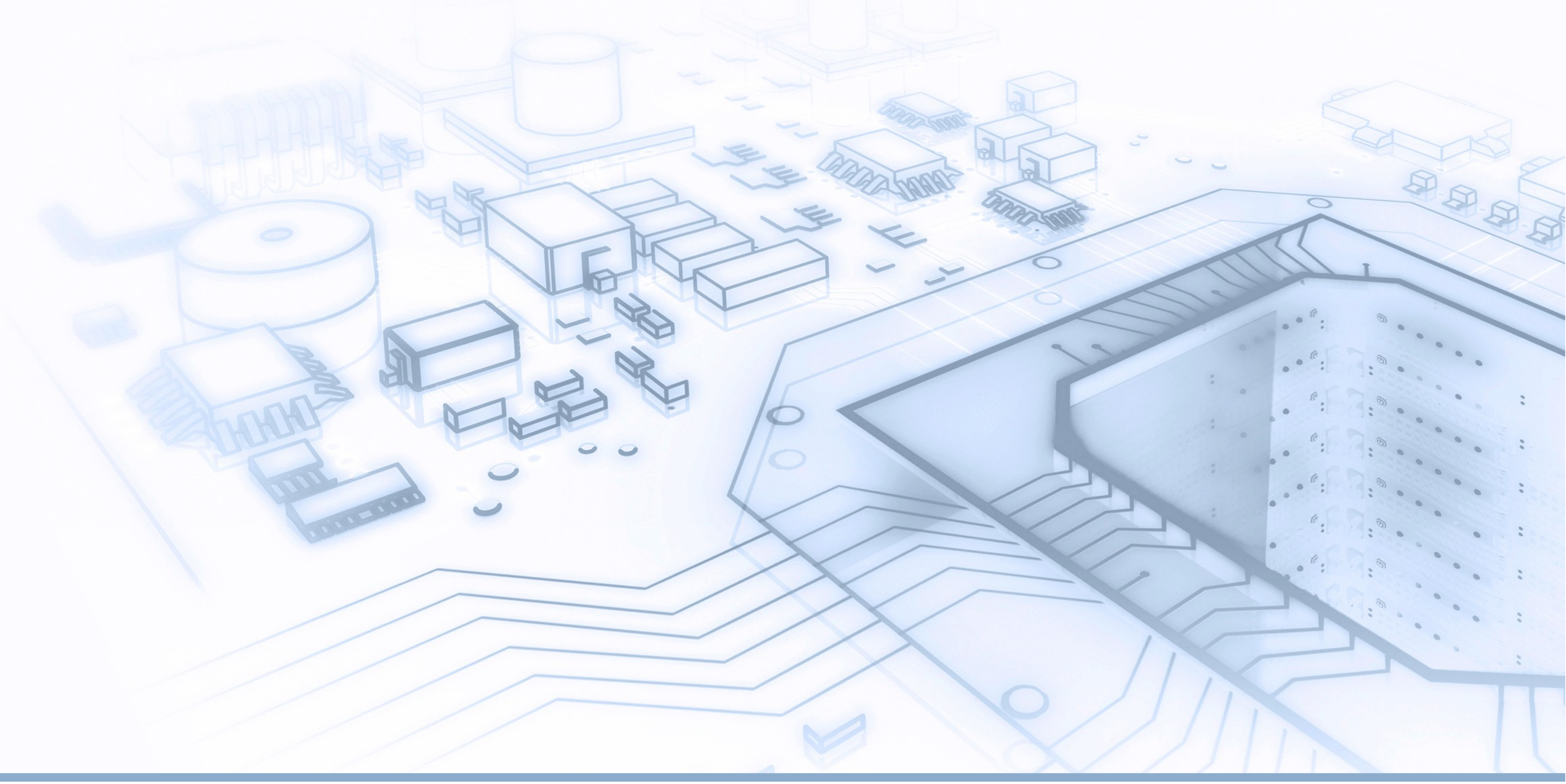
- Organization of iterations-threads mapping:
  - All threads perform the same number of iterations
  - Threads with sequential indexes perform the sequential iterations
    - ➔ Coalescing memory transactions
  - Support for mapping iterations space onto any compute grid space (extra inner loops)
- Recursive analysis of nested loops:
  - Utilize GPU capabilities for multidimensional compute grids creation
  - The most inner loop always correspond to the “x” axis of the GPU compute grid

Computing the optimal GPU compute grid

- After substituting the kernel arguments the loops dimensions become known
- With known loops dimensions it is possible to compute the optimal compute grid configuration:
  - ➔ Minimize the number of unused threads
  - ➔ Specify the number of iterations to perform in each thread

Runtime-optimization of loops' kernels

- Initially the “universal” LLVM IR is generated for kernel loops: one thread may perform more than one loop iteration
- Upon the kernel launch, the used compute grid becomes known
  - ➔ Extra IR-code optimizations could be performed after substituting the known constants
  - ➔ Reduce the register footprint
  - ➔ Eliminate fictive loops, less branching



Development plan

Programming

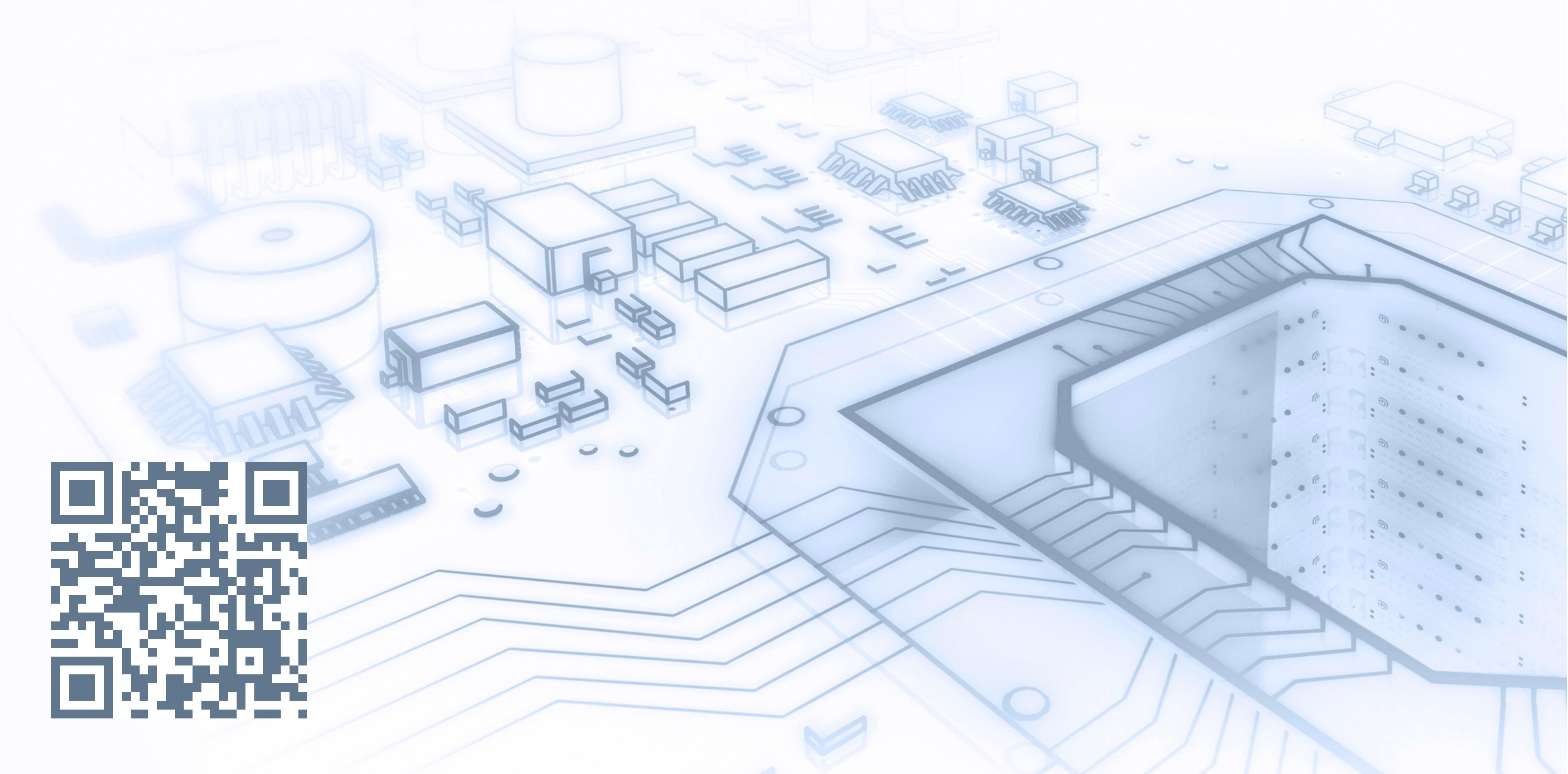
- Implement recursive nested loops analysis – already implemented during PCT conf ☺
- Privatize the global variables (no linker for GPU code)
- Re-implement host code JIT-compiler similar to lli
- Restructure the project source code (need better design)
- Implement kernels versions switcher heuristics and test it in real applications

Programming

- Optimizing data synchronization between CPU and GPU (#82, #84)
- Implement kernels configurations cache
- Implement application computational profile dumping for further reuse

Testing

- Test KernelGen on benchmarks and target applications:
  - Polybench, NPB, SPEC CPU 2006
  - COSMO, WRF, SLM, MF-RAPS
- Bug fixing

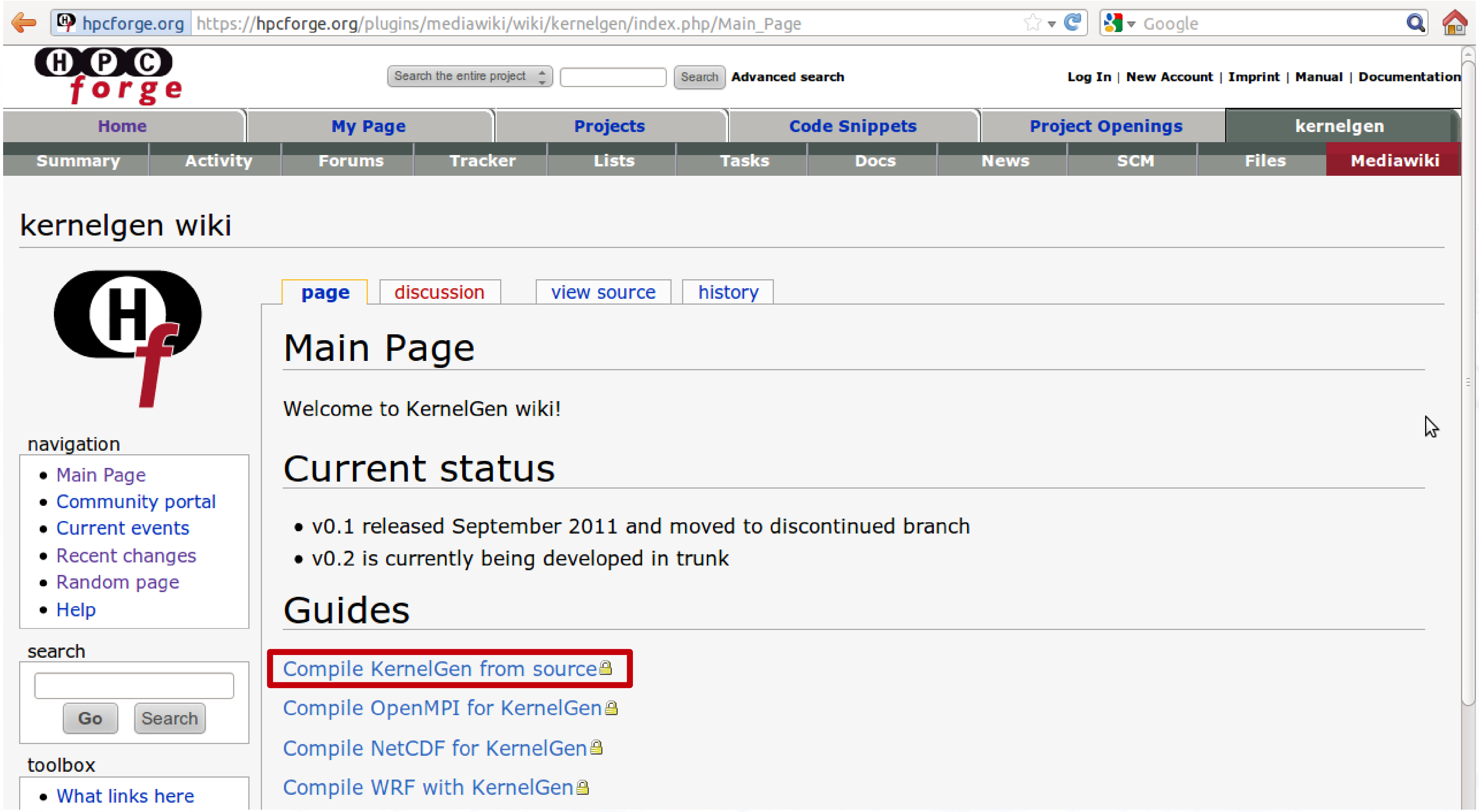


http://kernelgen.org

The work is supported by Applied Parallel Computing contracts 12-2011 and 13-2011, testing is performed on hardware installed at Lomonosov State University, SRCC MSU and supplied by NVIDIA.

Testing

- Instructions available at project wiki pages:



kernelgen wiki

Main Page

Welcome to KernelGen wiki!

Current status

- v0.1 released September 2011 and moved to discontinued branch
- v0.2 is currently being developed in trunk

Guides

Compile KernelGen from source

Compile OpenMPI for KernelGen

Compile NetCDF for KernelGen

Compile WRF with KernelGen

Resources

- [The LLVM Compiler Infrastructure]
- [Polly: Polyhedral optimizations for LLVM]
- [DragonEgg - Using LLVM as a GCC backend]