OpenCL Kernel Compilation

Shipping OpenCL Kernels

- OpenCL applications rely on ***online**** compilation in order to achieve portability
  - Also called runtime or JIT compilation
- Shipping source code with applications can be an issue for commercial users of OpenCL
- There are a few ways to try protect your OpenCL kernels

* OpenCL 2.2 C++ kernels are offline compiled – more later

Encrypting OpenCL Source

- One approach is to encrypt the OpenCL source, and decrypt it at runtime just before passing it to the OpenCL driver
- This could achieved with a standard encryption library, or by applying a simple transformation such as Base64 encoding
- This prevents the source from being easily read, but it can still be retrieved by intercepting the call to `clCreateProgramWithSource()`
- Obfuscation could also be used to make it more difficult to extract useful information from the plain OpenCL kernel source

Precompiling OpenCL Kernels

- OpenCL allows you to retrieve a binary from the runtime after it is compiled, and use this instead of loading a program from source
- This means that we can precompile our OpenCL kernels and ship the binaries with our application (instead of the source code)

Precompiling OpenCL Kernels

- **Retrieving the binary:**

```
// Create and compile program
program = clCreateProgramWithSource(context, 1, &kernel_source, NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// Get compiled binary from runtime
size_t size;
clGetProgramInfo(program, CL_PROGRAM_BINARY_SIZES, sizeof(size_t), &size, NULL);
unsigned char *binaries = malloc(sizeof(unsigned char) * size);
clGetProgramInfo(program, CL_PROGRAM_BINARIES, size, &binaries, NULL);

// Then write binary to file
…
```

- **Loading the binary**

```
// Load compiled program binary from file
…

// Create program using binary
program = clCreateProgramWithBinary(context, 1, devices, &size, &binaries,NULL,NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
```

Precompiling OpenCL Kernels

- These binaries are ***only*** valid on the devices for which they are compiled, so we potentially have to perform this compilation for ***every*** device we wish to target
- A vendor might change the binary definition at any time, potentially **breaking** our shipped application
- **If a binary isn't compatible** with the target device, **an error will be returned** either when creating the program or building it

Portable Binaries

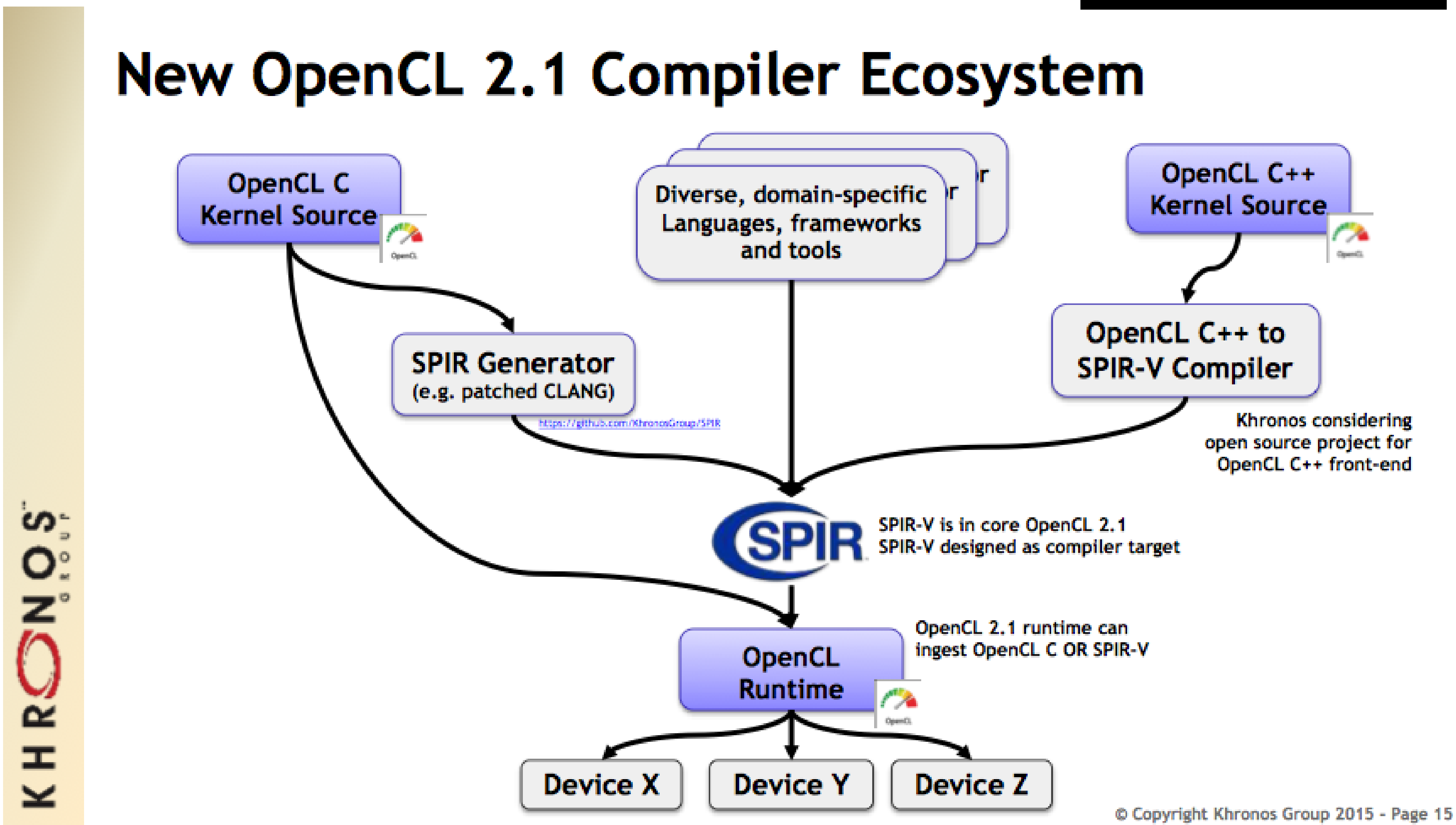
- Khronos has produced a specification for a **S**tandard **P**ortable **I**ntermediate **R**epresentation
- This defines a binary format that is designed to be portable, allowing us to use the same binary across many platforms
- Not yet supported by all vendors, but SPIR-V is now core from OpenCL 2.1 onwards
  - **`clCreateProgramWithIL()`**

SPIR-V Overview

- Cross-vendor intermediate language
- Supported as core by both OpenCL and Vulkan APIs
  - Two different 'flavors' of SPIR-V
  - Environment specifications describe which features supported by each
- Clean-sheet design, no dependency on LLVM
  - Open-source tools* provided for SPIR-V<->LLVM translation
- Enables alternative kernel programming languages
  - OpenCL 2.2 introduces a C++ kernel language using SPIR-V 1.2
- Offline compilation workflow
  - Lowered to native ISA at runtime

*http://github.khronos.org

SPIR-V Ecosystem



(IWOCL 2015, Stanford University)

Generating Assembly Code

- It can be useful to inspect compiler output to see if the compiler is doing what you think it's doing
- On NVIDIA platforms the 'binary' retrieved is actually PTX, their abstract assembly language
- On AMD platforms you can add **`-save-temps`** to the build options to generate **`.il`** and **`.isa`** files containing the intermediate representation and native assembly code
- Other vendors (such as Intel) may provide an offline compiler which can generate LLVM/SPIR or assembly

Kernel Introspection

- A mechanism for automatically discovering and using new kernels, without having to write any new host code
- This can make it much easier to add new kernels to an existing application
- Provides a means for libraries and frameworks to accept additional kernels from third parties

Kernel Introspection

- We can **query a program object** for the names of all the kernels that it contains:

```
clGetProgramInfo(program,CL_PROGRAM_NUM_KERNELS, …);
clGetProgramInfo(program,CL_PROGRAM_KERNEL_NAMES, …);
```

- We can also **query information about kernel arguments** (from OpenCL 1.2 onwards):

```
clGetKernelInfo(kernel, CL_KERNEL_NUM_ARGS, …);
clGetKernelInfo(kernel, CL_KERNEL_ARG_*, …);
```

(the program should be compiled using the `-cl-kernel-arg-info` option)

Separate Compilation and Linking

- OpenCL 1.2 gives more control over the build process by adding two new functions:

```
clCompileProgram(programs[0], …);
program = clLinkProgram(context,…,programs);
```

- This enables the creation of libraries of compiled OpenCL functions, that can be linked to multiple program objects
- Can improve program build times, by allowing code shared across multiple programs to be extracted into a common library

OpenCL Kernel Compiler Flags

- OpenCL kernel compilers accept a number of flags that affect how kernels are compiled:

```
-cl-opt-disable
-cl-single-precision-constant
-cl-denorms-are-zero
-cl-fp32-correctly-rounded-divide-sqrt
-cl-mad-enable
-cl-no-signed-zeros
-cl-unsafe-math-optimizations
-cl-finite-math-only
-cl-fast-relaxed-math
```

implies

OpenCL Kernel Compiler Flags

- Vendors may expose additional flags to give further control over program compilation, but these will not be portable between different OpenCL platforms
- For example, NVIDIA provide the **`-cl-nv-arch`** flag to specify which GPU architecture should be targeted, and **`-cl-nv-maxrregcount`** to limit the number of registers used
- Some vendors support **`-On`** flags to control the optimization level
- AMD allow additional build options to be dynamically added using an environment variable: **`AMD_OCL_BUILD_OPTIONS_APPEND`**

Other compilation hints

- Can use an attribute to inform the compiler of the work-group size that you intend to launch kernels with:

  `__attribute__((reqd_work_group_size(x, y, z)))`

- As with C/C++, use the **`const`**/**`restrict`** keywords for kernel arguments where appropriate to make sure the compiler can optimise memory accesses

Metaprogramming

- We can exploit runtime kernel compilation to embed values that are only known at runtime into kernels as compile-time constants
- In some cases this can significantly improve performance
- OpenCL compilers support the same preprocessor definition flags as GCC/Clang:

```
-Dname
-Dname=value
```

Example: Multiply a vector by a constant value

Passing the value as an argument

```
kernel void vecmul(
  global float *data,
  const  float  factor)
{
  int i = get_global_id(0);
  data[i] *= factor;
}
```

Value of 'factor' not known at application build time (e.g. passed as a command-line argument)

```
clBuildProgram(program, 0, NULL, NULL,
NULL, NULL);
```

Example: Multiply a vector by a constant value

Passing the value as an argument

```
kernel void vecmul(
  global float *data,
  const  float  factor)
{
  int i = get_global_id(0);
  data[i] *= factor;
}
```

```
clBuildProgram(program, 0, NULL,
NULL, NULL, NULL);
```

Defining the value as a preprocessor macro

```
kernel void vecmul(
  global float *data)

{
  int i = get_global_id(0);
  data[i] *= factor;
}
```

```
sprintf(options, "-Dfactor=%f",
userFactor);
```

```
clBuildProgram(program, 0, NULL,
options, NULL, NULL);
```

Metaprogramming

- Can be used to dynamically change the precision of a kernel
  - Use **`REAL`** instead of **`float/double`**, then define **`REAL`** at runtime using OpenCL build options: **`-DREAL=type`**
- Can make runtime decisions that change the functionality of the kernel, or change the way that it is implemented to improve performance portability
  - Switching between scalar and vector types
  - Changing whether data is stored in buffers or images
  - Toggling use of local memory

Metaprogramming

- All of this requires that we are compiling our OpenCL sources at runtime – this doesn't work if we are precompiling our kernels or using SPIR
- OpenCL 2.2 and SPIR-V provide the concept of *specialization constants*, which allow symbolic values to be set at runtime

```
// OpenCL C++ kernel code
// Create specialization constant with ID 1 and default value of 3.0f
cl::spec_constant<float, 1> factor = {3.0f};
data[i] *= factor.get();

// Host code
// Set value of specialization constant and then build program
cl_uint spec_id = 1;
clSetProgramSpecializationConstant(program, spec_id,
                                   sizeof(float), &userFactor);
clBuildProgram(program, 1, &device, "", NULL, NULL);
```

Auto tuning

- Q: How do you know what the ***best*** parameter values for your program are?
  - What is the best work-group size, for example?

- A: Try them all! (Or a well chosen subset)

- This is where auto tuning comes in
  - Run through different combinations of parameter values and optimize the runtime (or another measure) of your program.

Tuning Knobs: Some general issues to think about

- Tiling size (work-group sizes, dimensionality etc.)
  - For block-based algorithms (e.g. matrix multiplication)
  - Different devices might run faster on different block sizes
- Data layout
  - Array of Structures or Structure of Arrays (AoS vs. SoA)
  - Column or Row major
- Caching and prefetching
  - Use of local memory or not
  - Extra loads and stores assist hardware cache?
- Work-item / work-group data mapping
  - Related to data layout
  - Also how you parallelize the work
- Operation-specific tuning
  - Specific hardware differences
  - Built-in trig / special function hardware
  - Double vs. float (vs. half)

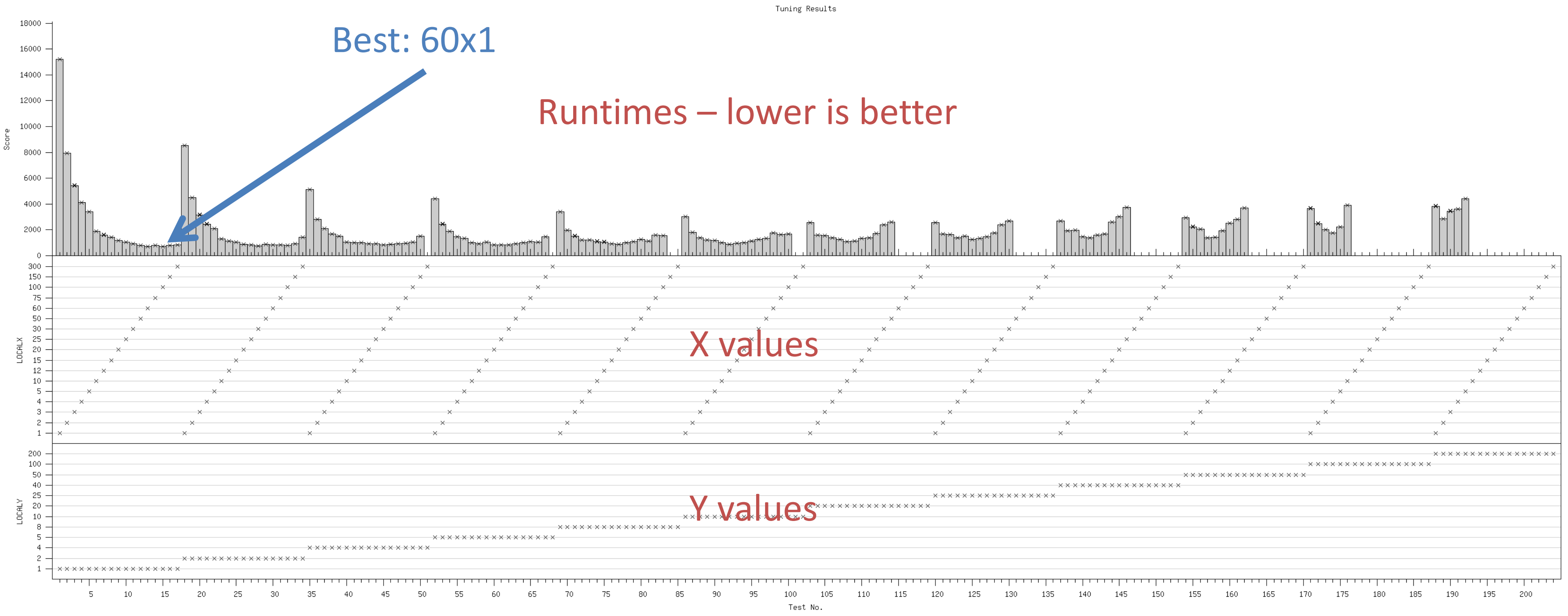
From Zhang, Sinclair II and Chien: Improving Performance Portability in OpenCL Programs – ISC13

Auto tuning example - Flamingo

- http://mistymountain.co.uk/flamingo/
- Python program which compiles your code with different parameter values, and calculates the “best” combination to use
- Write a simple config file, and Flamingo will run your program with different values, and returns the best combination
- Remember: scale down your problem so you don’t have to wait for “bad” values (less iterations, etc.)

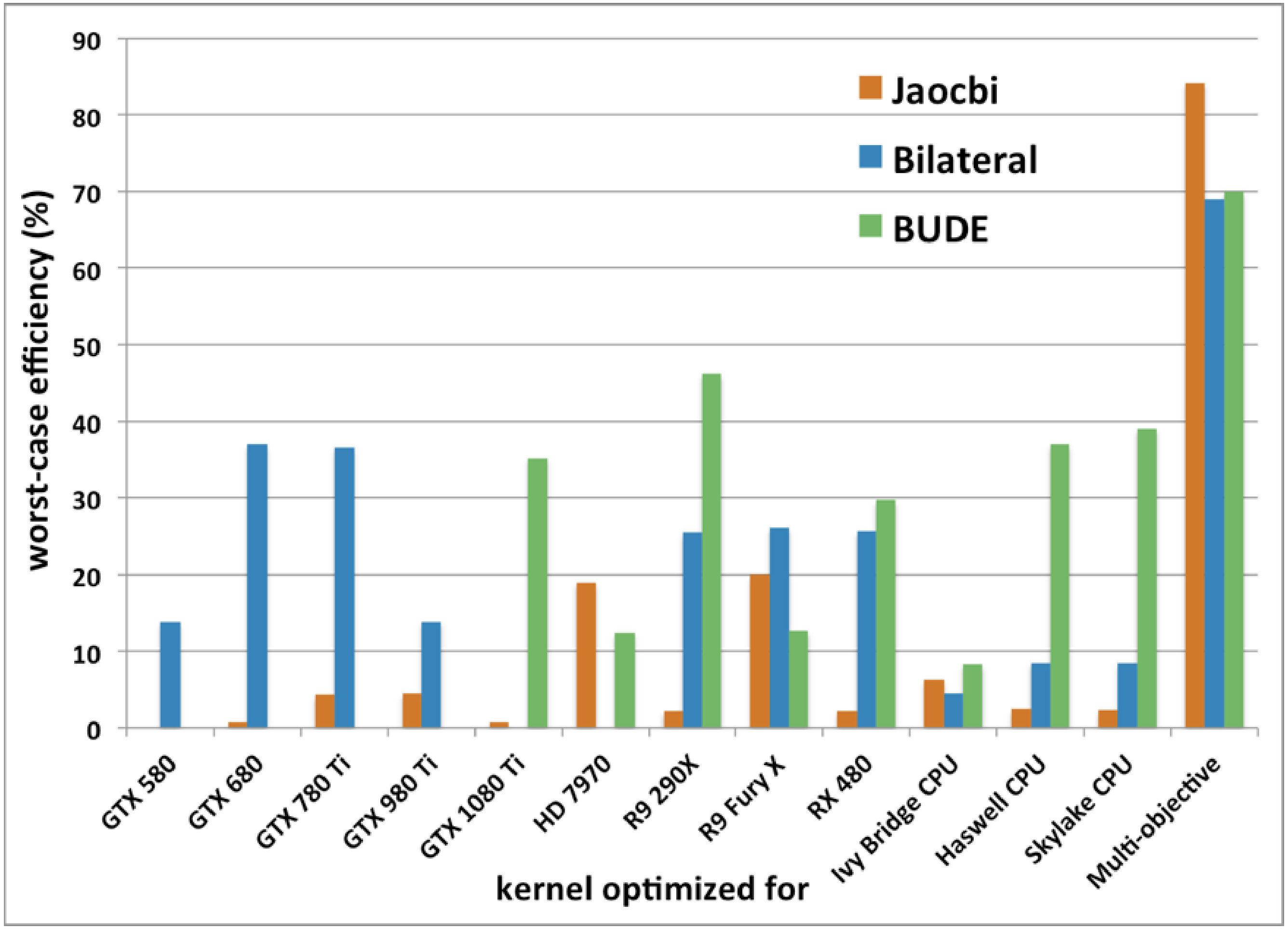
Auto tuning - Example

- D2Q9 Lattice-Boltzmann
- What is the best work-group size for a specific problem size (3000x2000) on a specific device (NVIDIA Tesla M2050)?



Collected with Flamingo (mistymountain.co.uk/flamingo)

Multi-objective auto-tuning (IWOCL'17)



"Analyzing and improving performance portability of OpenCL applications via auto-tuning", J.Price and S.McIntosh-Smith, IWOCL 2017, https://dl.acm.org/citation.cfm?id=3078173