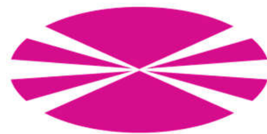
Facilitating the Programming of Heterogeneous Devices using the HPL Library

Diego Andrade

(diego.andrade@udc.es)



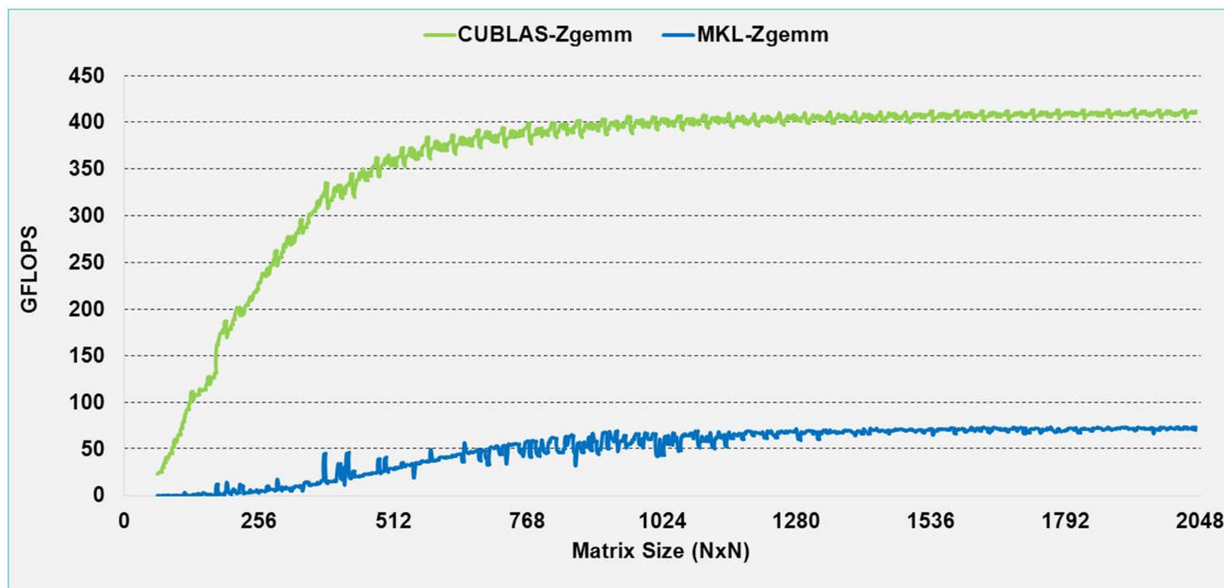
UNIVERSIDADE DA CORUÑA

About me

- My PH.D thesis versed about predicting the cache performance of sparse codes
- After that:
  - Predicted the cache performance of applications with application with soft and hard real-time constraints
  - Predicted the cache performance of multicore cache hierarchies with private and shared cache levels
- Lately:
  - Providing performance portability on heterogeneous systems, using:
    - OCLoptimizer: a source to source optimization tool built on top of CLANG
    - With HPL

Computing landscape

- Heterogeneous systems are being increasingly adopted
  - Large performance and power benefits
  - No single device is the best for everything
  - Most code is (yet) CPU-only



- cuBLAS 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

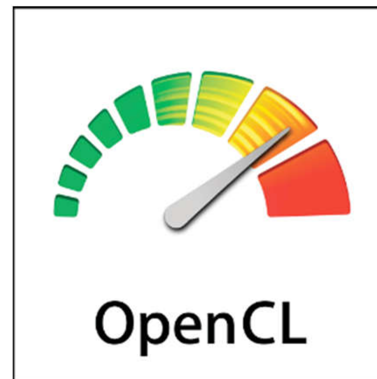
•Performance may vary based on OS ver. and motherboard config.

Problems of heterogeneity

- More and more different systems
- Greater programming effort
  - New software managed hardware features
  - Separate memory: communication with host CPU
- Portability can be compromised
  - Each vendor/device may have its development environment. E.g.: CUDA

OpenCL

- Open standard for general-purpose programming of heterogeneous systems
  - Since 2008
- Improves the situation but
  - Complex programming interface
    - Many small steps, low level management
  - Lacks features (e.g. templates until V2.2)



How can we improve this?

- Several libraries have tried to address this problem with different degrees of
  - Restrictions on expressivity
  - Portability
  - Exposure of the underlying hardware or specific language
- Compiler directives
  - Sometimes, problems similar to libraries
  - Dependent on compiler technology

Purpose

- Library-based solution to program heterogeneous systems
  - Expressive
  - Easy to use
  - Portable
  - No/minimal need to learn new languages
  - Good performance

Heterogeneous Programming Library (HPL)

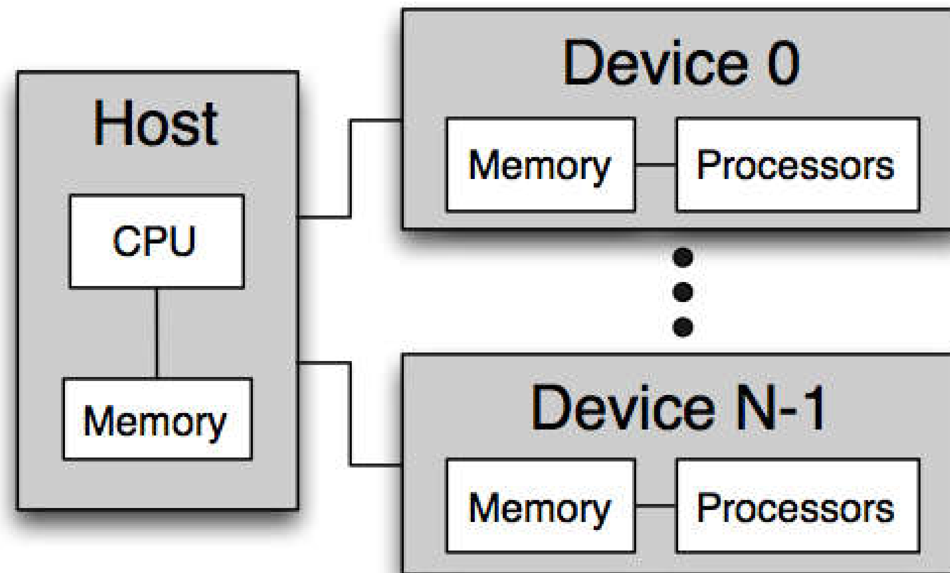
- C++ library based on two key concepts
  - Kernels: functions that are evaluated in parallel by multiple threads on any device
  - Data types to express arrays and scalars that can be used in kernels and serial code
- Low level code generation at runtime
  - Easies specialization, code space search

Download it at: http://hpl.des.udc.es

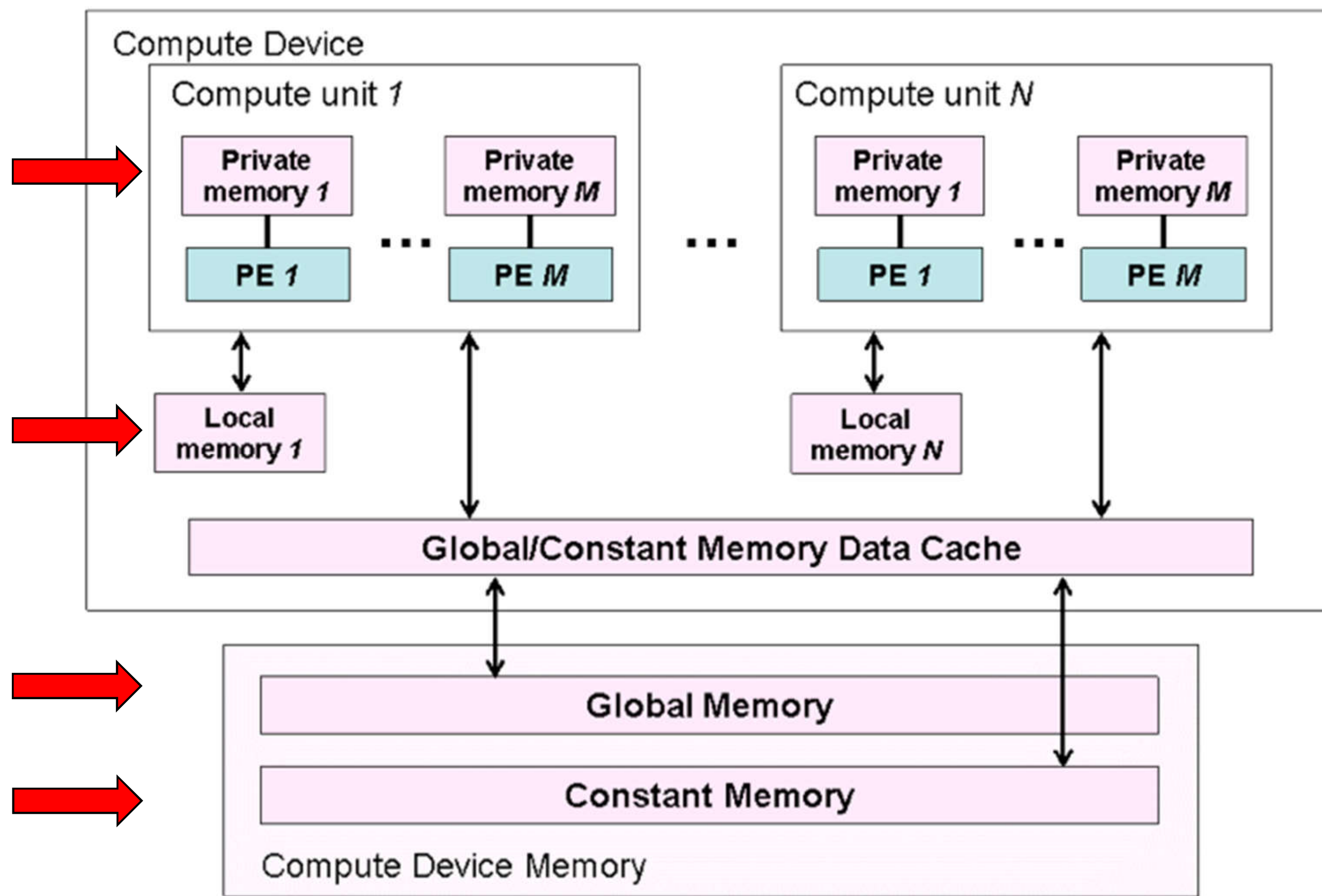
HPL Description

HPL hardware model

- Serial code runs in the host
- Parallel kernels can be run everywhere
- Processors can only access their device memory

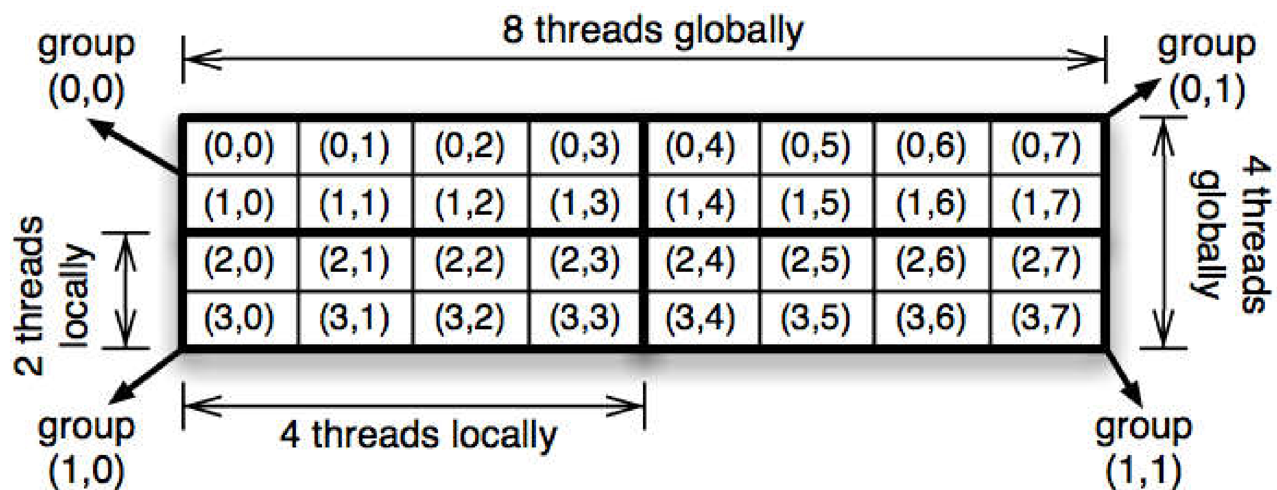


Supports the whole OpenCL memory model



Kernel evaluation index space

- Global domain required
  - Provides unique ID for each parallel thread
- Optional local domain
  - Threads in the same local domain can share local memory and synchronize with barriers



Example: SAXPY (Y=*a**X+Y)

```
void saxpy(Array<float, 1> y, Array<float, 1> x, Float a, Int n)
{
  if_(idx < n) {
    y[idx] = a * x[idx] + y[idx];
  }
}

int main()
{
  int n = . . .; // length of the arrays
  float a = . . .;

  Array<float, 1> x(n), y(n);

  //the vectors are filled in with data (not shown)
  int nGlobalThreads = ceil(n / 32.) * 32;

  eval(saxpy).global(nGlobalThreads).local(32)(y, x, a, n);
}
```

Idx: thread global id in the first dimension

Arrays

- **Array**<type, ndims [,memoryFlag]> definition of an array that can be used in host code and kernels
  - Examples:
    - Array<float, 2> matrixX(100, 100);
    - Array<int,1,Local> arrayX(500);
    - Array<int,0> scalar;
  - memoryFlag: **Global**, Local, Private or Constant
    - Kernel arguments default to global memory
    - Kernel in-function variables default to private memory

Kernels

- Anything executable in C++
  - Regular functions, functors, etc.
  - Single-source heterogeneus computing!
  - Written with HPL kernel language
- The arguments are the kernel arguments
  - Scalars are passed by value
  - Arrays are passed by reference

HPL kernel language

- Control flow structs with underscore:
  - if ⇨ **if_**; else ⇨ **else_**; etc.
  - for ⇨ **for_** with commas separating the arguments
- Predefined variables to identify threads, get number of threads, etc.
- Functions for mathematical operations, synchronizations, etc.

Host API: Running kernels

- **eval**(kernel)(args) parallel evaluation of kernel on the arguments
- Default global domain = size of the first argument
  - Can be specified with **global(x,y,z)**
- Default local domain chosen by library
  - Can be specified with **local(x,y,z)**
- Execution takes place by default in the first accelerator found in the system, or otherwise the CPU if it supports OpenCL
  - Can be specified with **device(d)**
- Functions to inspect existing devices and their properties

Example: Templates for heterogeneous systems

```
template<typename T>
void add(Array<T, 2> a, Array<T, 2> b,
      Array<T, 2> c) {
  a[idx][idy] =  b[idx][idy] + c[idx][idy];
}

. . .

Array<float, 2> av(N,N), bv(N,N), cv(N,N);
Array<int, 2> avi(M,M), bvi(M,M), cvi(M,M);

//We use addv to add floats
eval(addv<float>)(cv, av, bv);

//We use addv to add ints
eval(addv<int>)(cvi, avi, bvi);
```

Kernel execution semantics

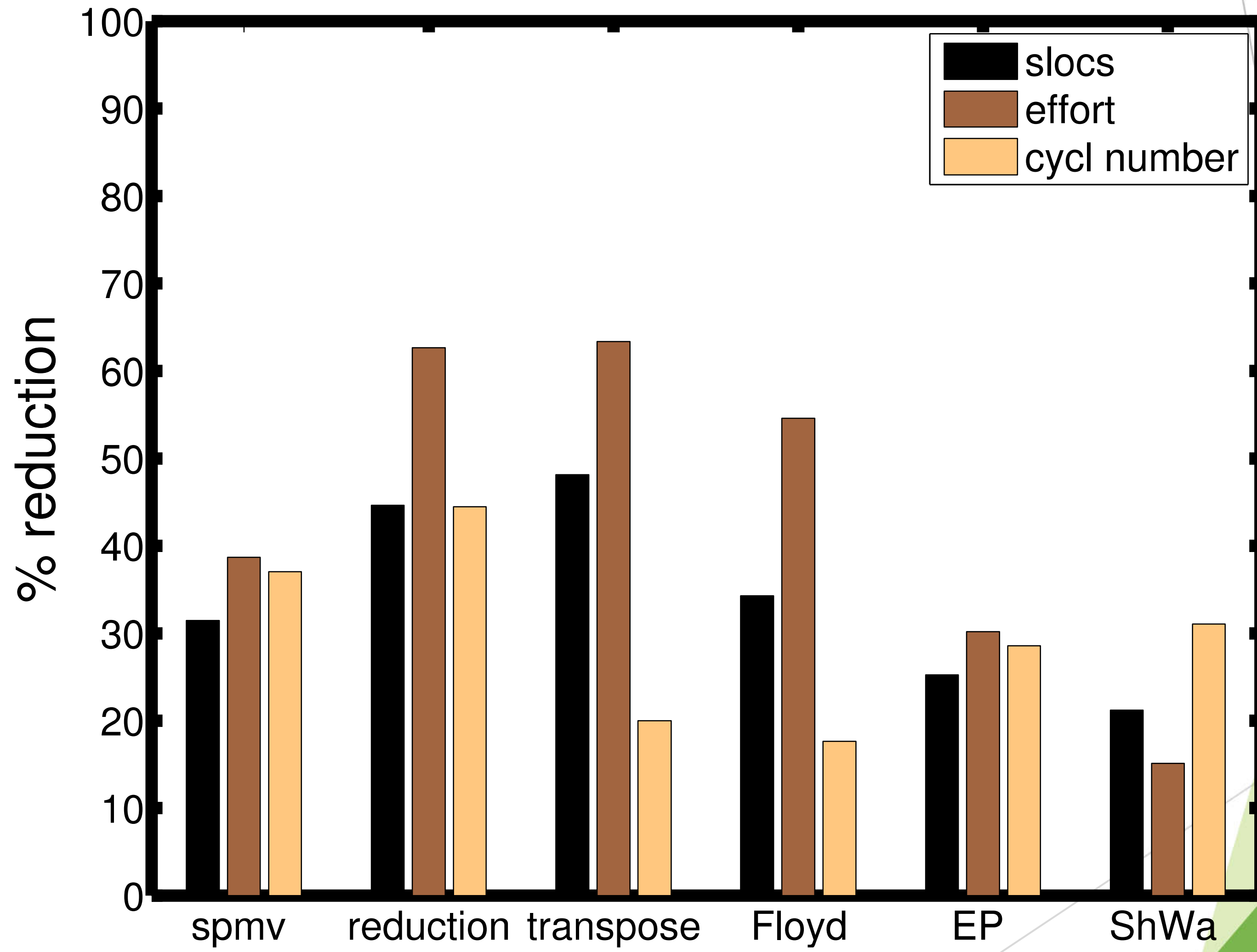
- Kernel evaluations are asynchronous
  - Host continues executing in parallel with the device
  - Facilitate overlapped operation host/device
  - Easy exploitation of multiple devices
- Synchronization based on the accesses to data
- Arrays are kept consistent across their usages both in different kernels and in the host
  - When a kernel needs data updated by another one, HPL automatically waits in order to provide the correct data
  - Sequential consistency provided

OpenCL generation

- The code is executed as regular C++
- HPL elements capture the code of the kernels
  - Allows to build a representation in OpenCL C
  - The code generated can be obtained
- Normal C++ can be mixed with HPL elements
  - Allows metaprogramming
- Simple analyses are performed
  - e.g.: which arrays are read, written or both
    - Enables automated & optimal management of array transfers

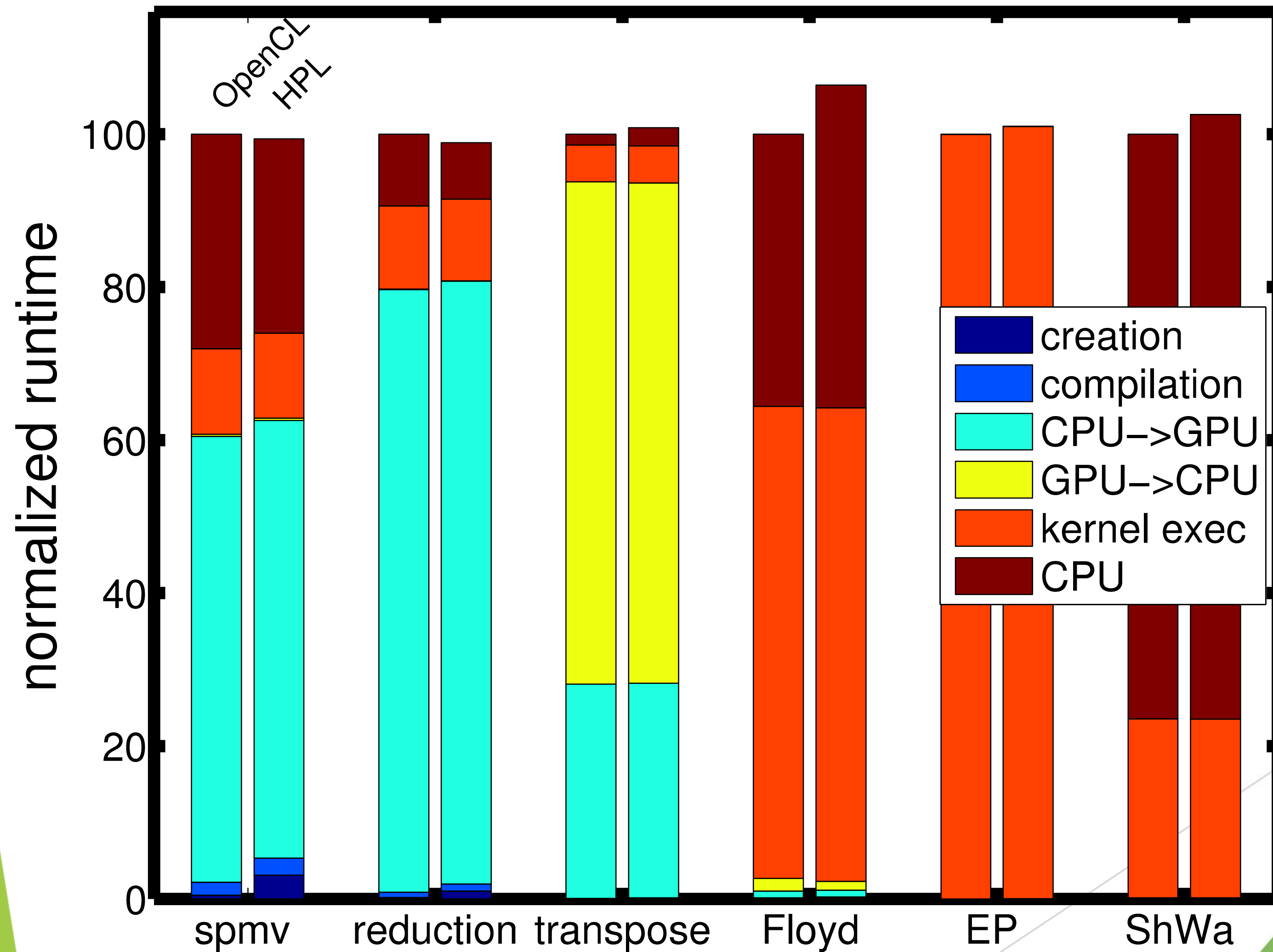
Evaluation

OpenCL VS HPL: Programmability - factorized

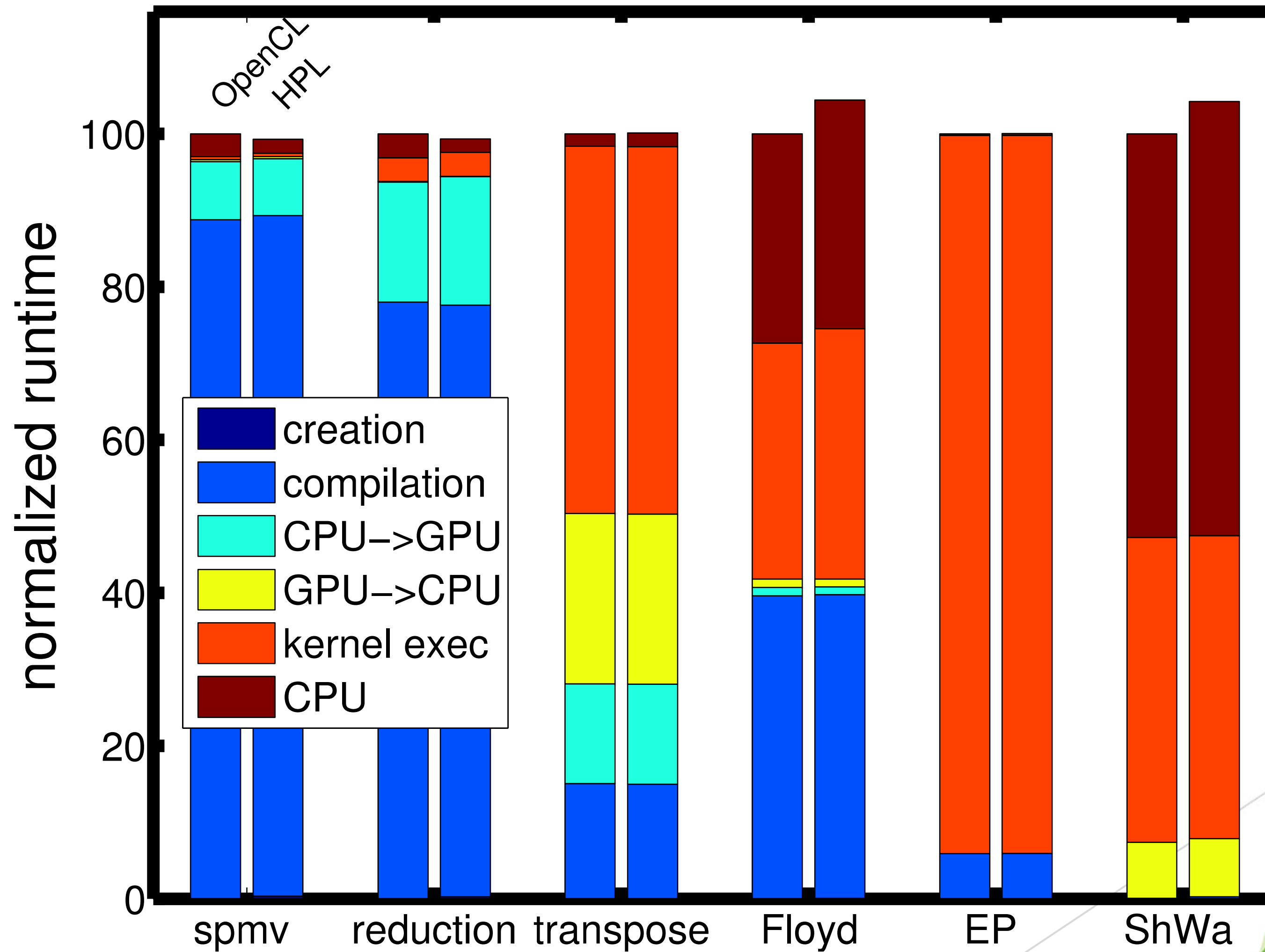


-270 SLOCs of initialization in the baseline

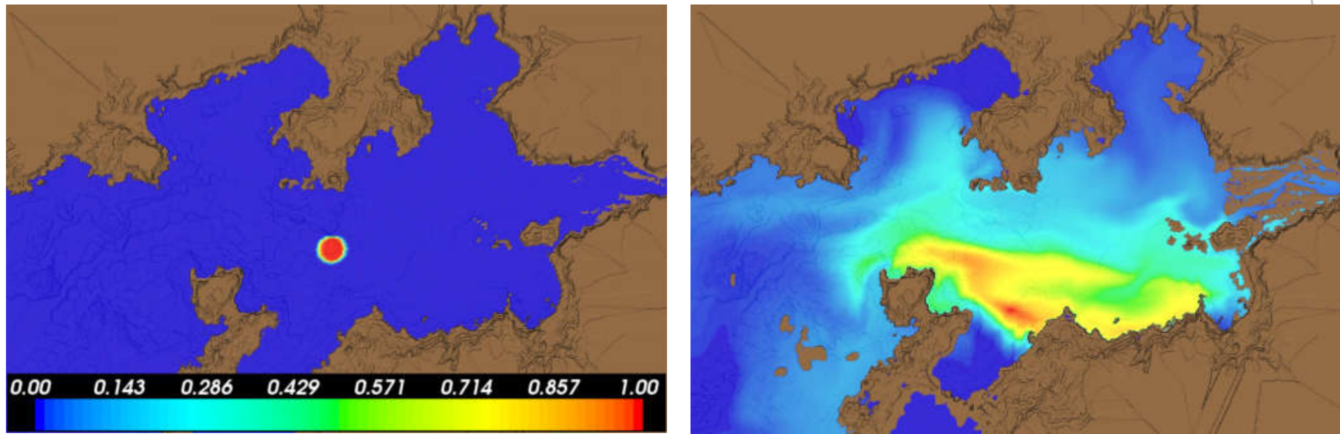
OpenCL – HPL performance tests in Tesla C 2050/C 2070



OpenCL – HPL performance tests in AMD HD6970

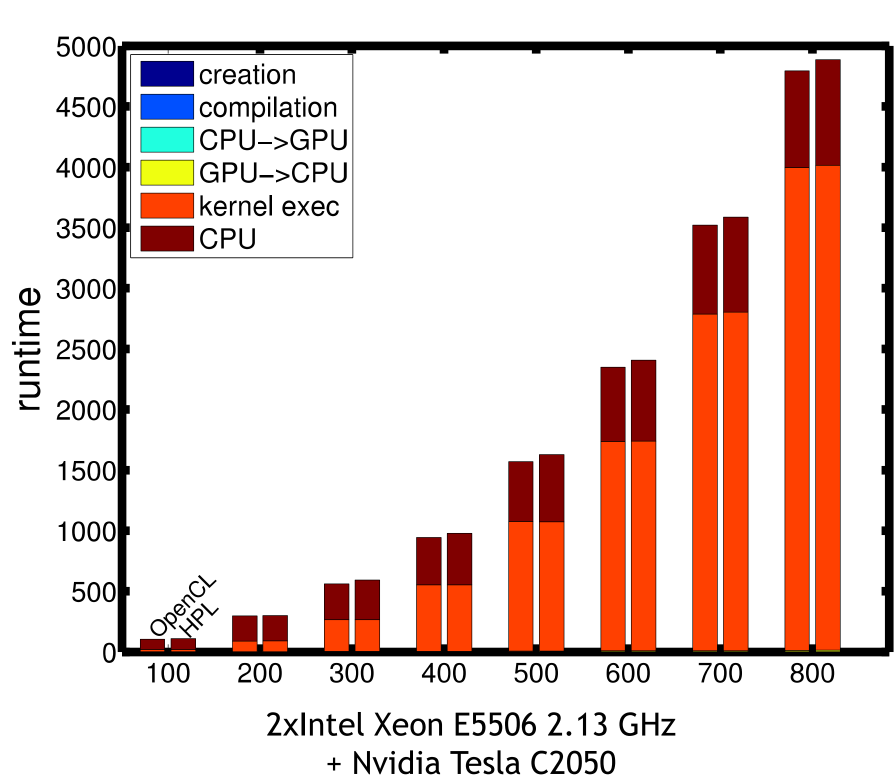


Real world app: Shallow water simulator with contaminants

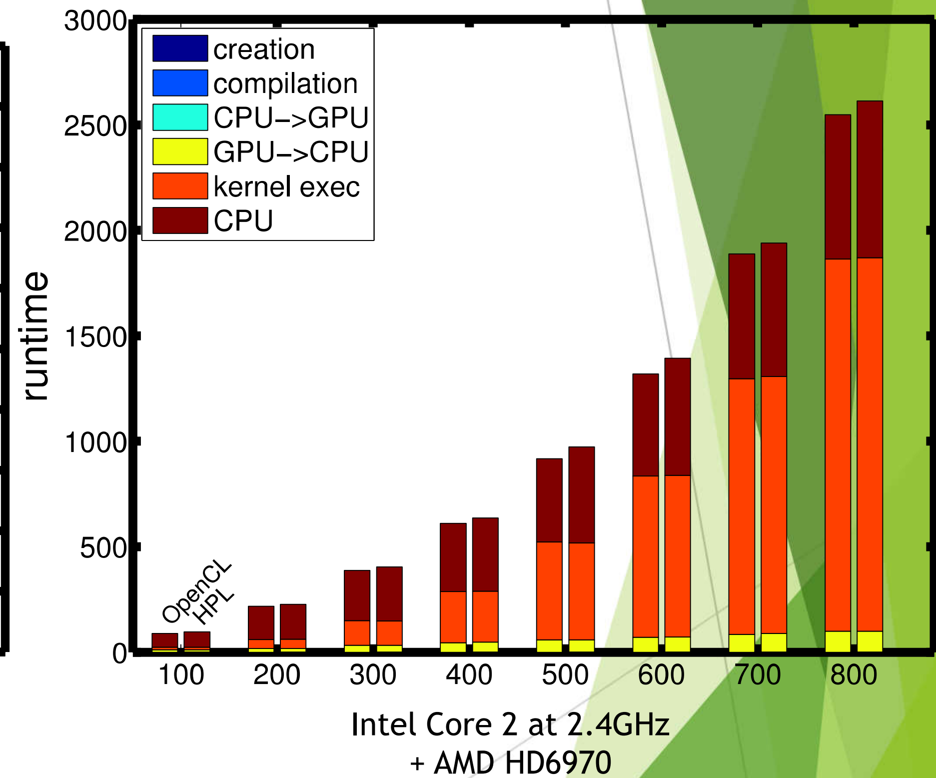


Prize from Fujitsu

Shallow water simulator performance



2xIntel Xeon E5506 2.13 GHz
+ Nvidia Tesla C2050



Intel Core 2 at 2.4GHz
+ AMD HD6970

Autotuning

Meta-programming

- Regular C++ can be interleaved in the kernels
  - C++ variables are frozen as constants
  - C++ statements are executed, not captured
    - They do not generate OpenCL code
    - But they can control the OpenCL code generated
  - Can manipulate at runtime the generation of code!

Example: metaprogramming

```
if( ( (m * (m + 1) ) / 2 ) * n > C ) {
   Int i, j, k;
   for_( i  = 0, i < m, i++ ) //generate for loops
      for_( j = 0, j < n, j++ )
         for_( k = i; k < m; k++ )
            r[i][j] += a[i][k] * b[k][j];
} else {
   for( int i = 0; i < m; i++ ) //generate full unroll
      for( int j = 0; j < n; j++ )
         for( int k = i; k < m; k++ )
           r[i][j] += a[i][k] * b[k][j];
 }
```

It generates...

```
for( i  = 0; i < m; i++ )
       for( j = 0; j < n;  j++ )
          for( k = i; k < m; k++ )
             r[i][j] += a[i][k] * b[k][j];
```

Or...

```
r[0][0] += a[0][0] * b[0][0];
r[0][0] += a[0][1] * b[1][0];
r[0][0] += a[0][2] * b[2][0];
…
```

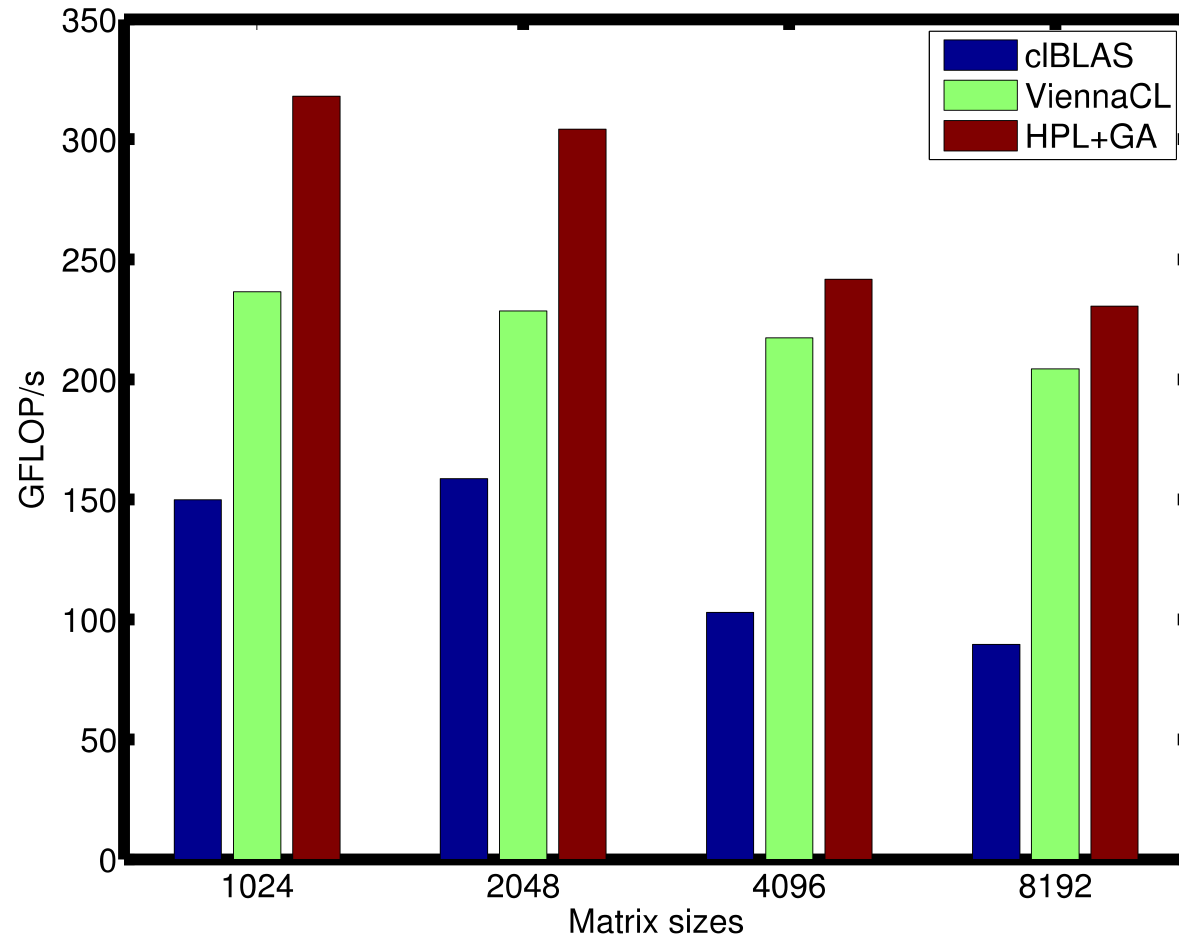
Allows control on

- Tiling
- Loop ordering
- Vectorization
- Usage of local memory
- Unroll
- Granularity
- …

Matrix product adaption using HPL + Genetic Algorithms

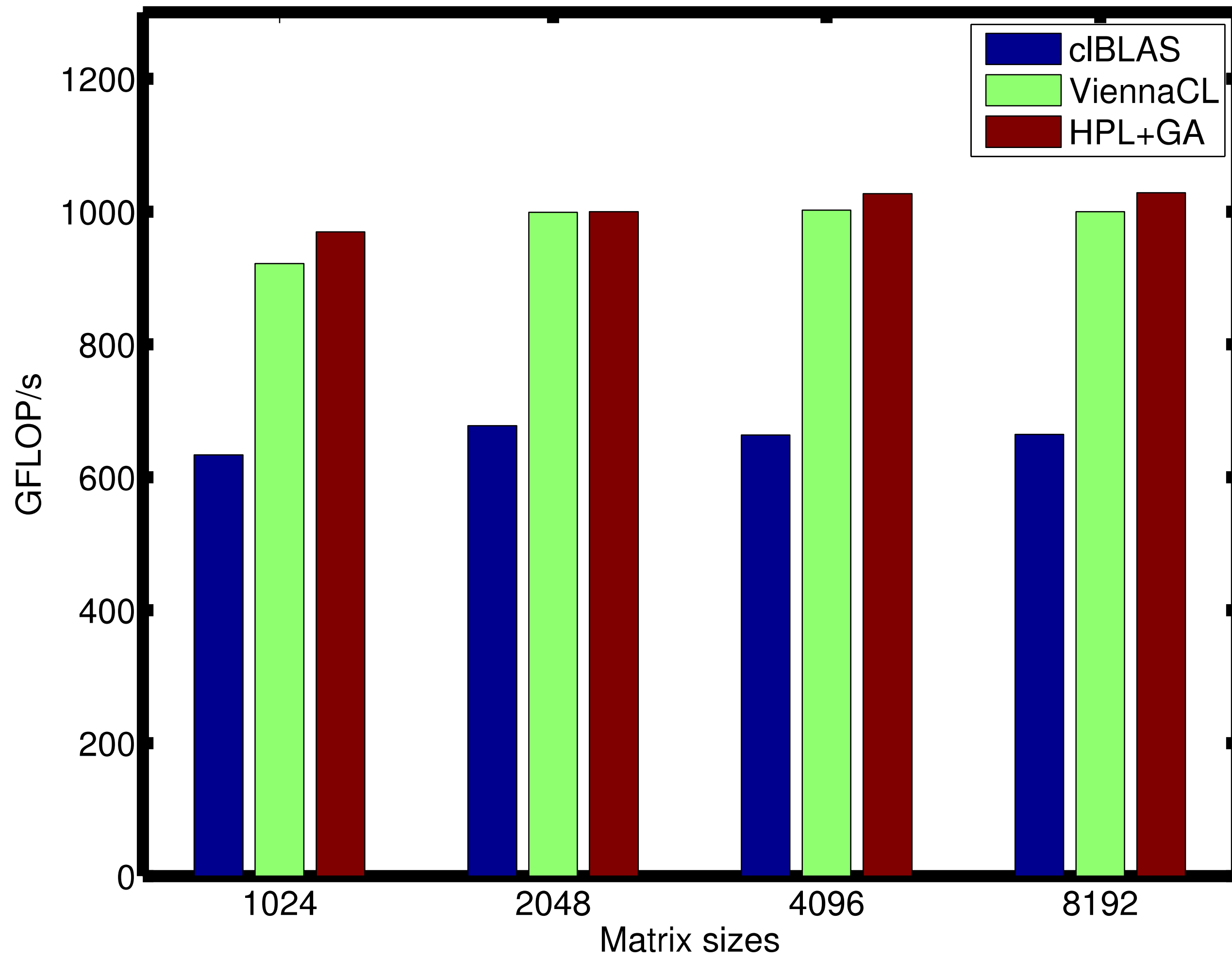
- Matrix product is one of the most widely used kernels
- We wrote a matrix product with adaptive features based on HPL run time code generation and metaprogramming
  - Optimization space searched with a genetic algorithm
- We compared with state of the art adaptive OpenCL BLAS libraries:
  - clBLAS from AMD
  - ViennaCL from TU Wien

Results in Intel CPU



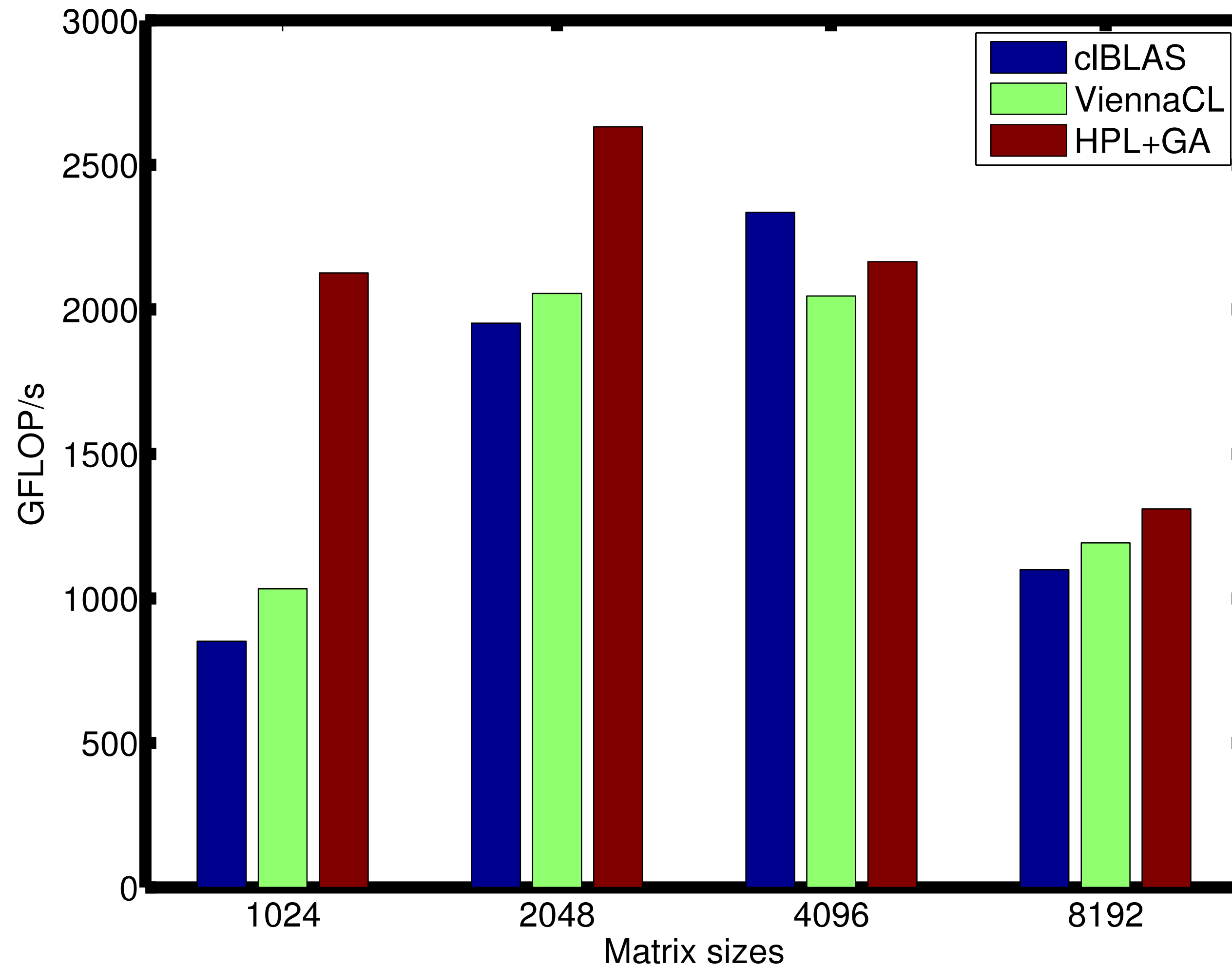
2x8-core Intel E5-2660 (2.2 GHz) CPU

Results in Nvidia GPU



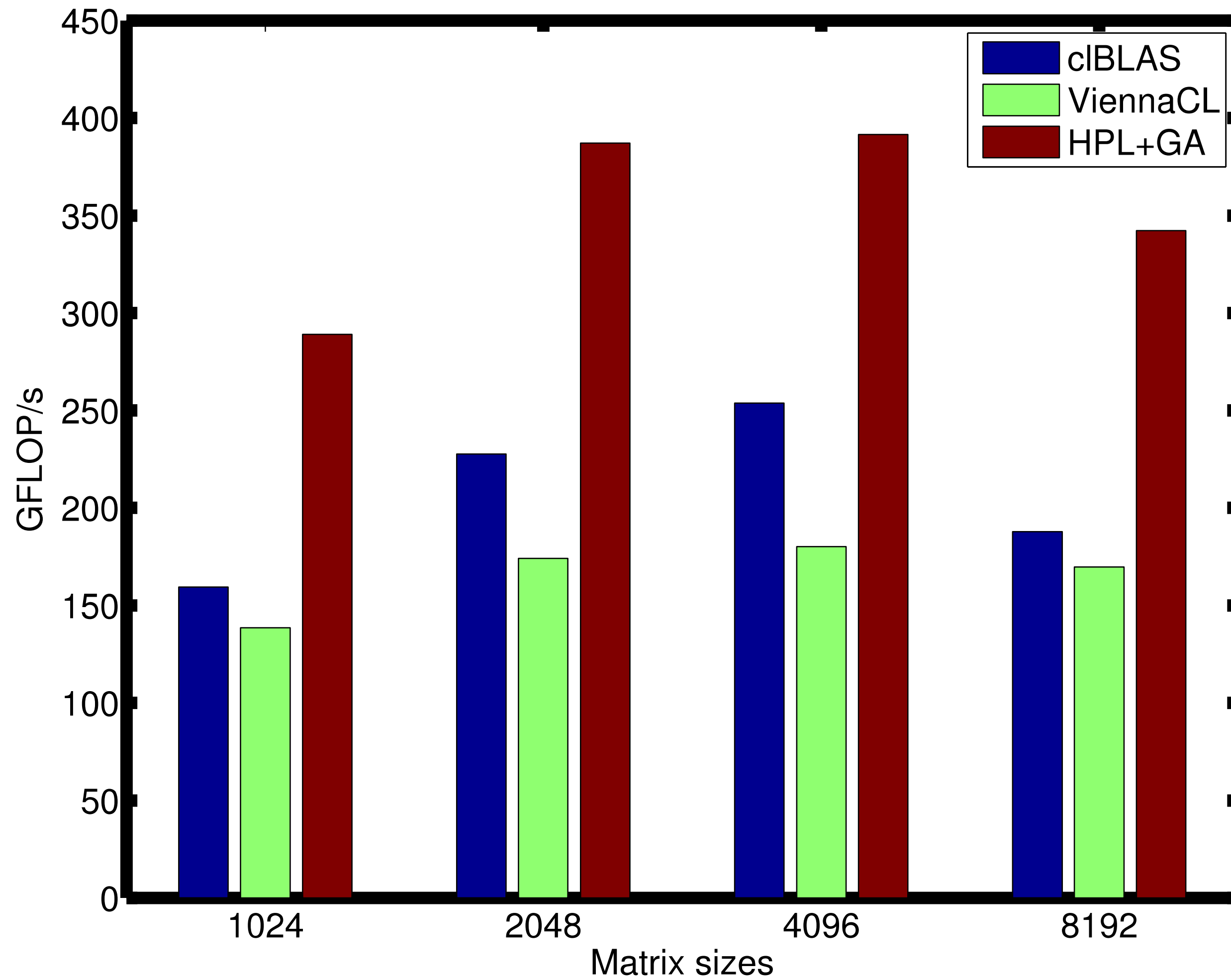
NVIDIA Tesla K20m with 2496 cores (705 MHz)

Results in AMD GPU



AMD FirePro S9150 with 2816 cores (900 MHz)

Results in Intel Xeon Phi



Intel Xeon Phi 5110P with 60 cores (1.053GHz)

Tuning times

Device	Size	Total tuning time (s)		
		GA	clBLAS	ViennaCL
CPU	1024	120.57	42947.26	32428.25
	2048	339.99		60438.13
	4096	1729.80		500775.18
	8192	19286.90		4186086.80
Nvidia	1024	242.04	1225.53	18836.30
	2048	331.40		38292.62
	4096	4429.57		186041.36
	8192	17127.50		1394675.71
AMD	1024	1579.74	5425.97	1911.00
	2048	2422.34		6221.00
	4096	4587.55		60595.37
	8192	5792.07		> 3 days
ACC	1024	260.32	86501.20	121891.58
	2048	915.69		211610.18
	4096	4401.47		1145630.97
	8192	31973.30		> 3 days

Conclusions

- HPL facilitates portable high performance programming of heterogeneous systems
- Large programmability improvements over OpenCL
- Typical performance overhead << 5%
- Good support for multi-device execution
- Success at generating optimized kernels
- Under GPL V3 at http://hpl.des.udc.es

Also available

- Easy-to-use multi-device support
  - With amenable methods to implement load-balancing support

Future & Ongoing Work

- Distributed memory systems / clusters
  - To be published soon.
- Facilitating code space exploration
  - Optimization abstractions
- Increased programmability within kernels

Thank you!

http://hpl.des.udc.es

diego.andrade@udc.es