Locality-Aware Automatic Parallelization for GPGPU with OpenHMPP Directives

José M. Andión, Manuel Arenaz, François Bodin, Gabriel Rodríguez and Juan Touriño

7th International Symposium on High-Level Parallel Programming and Applications (HLPP 2014)
July 3-4, 2014 — Amsterdam, Netherlands

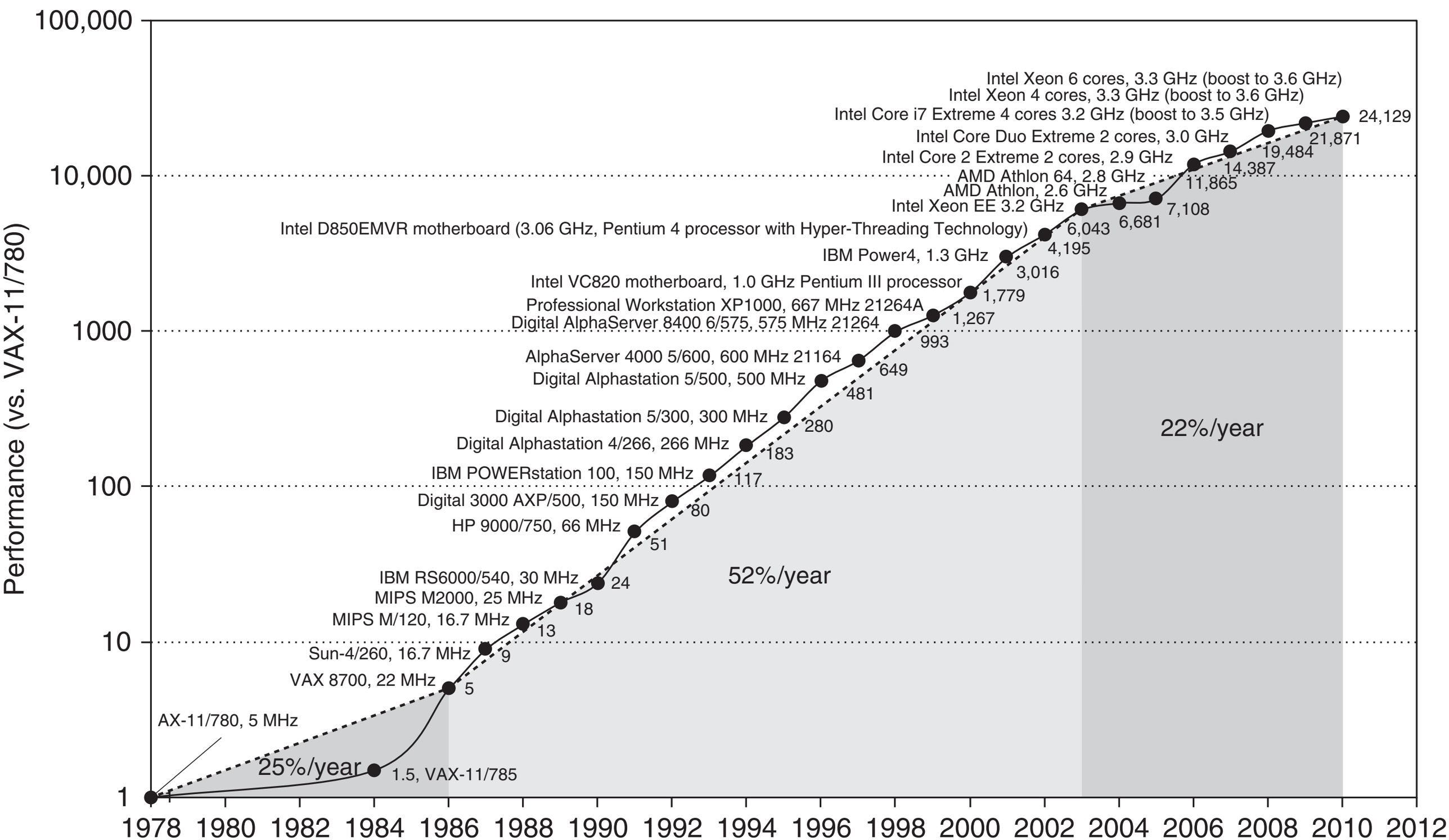
COMPUTER ARCHITECTURE GROUP
UNIVERSITY OF A CORUÑA
GAC.UDC.ES
UNIVERSIDADE DA CORUÑA

Outline

- Motivation: General Purpose Computation with GPUs
- GPGPU with CUDA & OpenHMPP
- The KIR: an IR for the Detection of Parallelism
- Locality-Aware Generation of Efficient GPGPU Code
- Case Studies: CONV3D & SGEMM
- Performance Evaluation
- Conclusions & Future Work

Outline

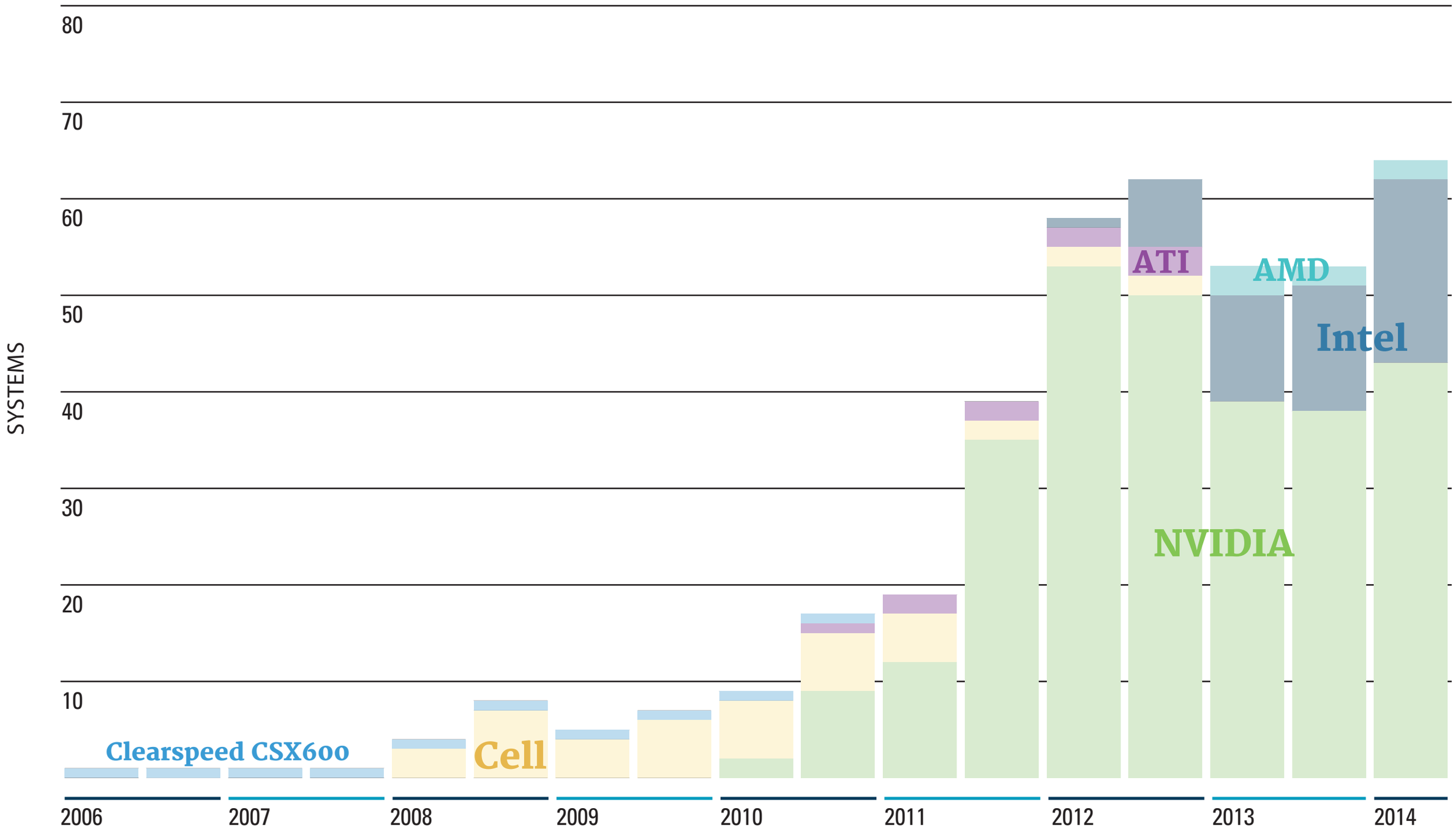
- Motivation: General Purpose Computation with GPUs
- GPGPU with CUDA & OpenHMPP
- The KIR: an IR for the Detection of Parallelism
- Locality-Aware Generation of Efficient GPGPU Code
- Case Studies: CONV3D & SGEMM
- Performance Evaluation
- Conclusions & Future Work



The Parallel Challenge

David A. Patterson and John L. Hennessy.
Computer Organization and Design:
The Hardware/Software Interface.
Elsevier, 2014.

ACCELERATORS/CO-PROCESSORS



General Purpose Computing with GPUs

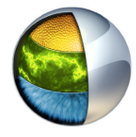
The TOP500 List.
June 2014.

Outline

- Motivation: General Purpose Computation with GPUs
- **GPGPU with CUDA & OpenHMPP**
- The KIR: an IR for the Detection of Parallelism
- Locality-Aware Generation of Efficient GPGPU Code
- Case Studies: CONV3D & SGEMM
- Performance Evaluation
- Conclusions & Future Work

GPGPU with CUDA

- First GPGPU programs look like graphics applications



- CUDA enables the use of C

CUDA kernel: specifies the operation of a single GPU thread

- Main ideas:

1. Lightweight parallel threads in hierarchy: grid, block
2. Shared memory
3. Barriers

GPU Programming Features in CUDA

1	Threadification
2	Thread grouping: *warps*
3	Minimization of CPU-GPU data transfers
4	Coalescing
5	Maximization of the usage of registers and shared memory
6	Divergency
7	Occupancy
8	Threads per block

GPGPU with OpenHMPP

- Directive-based approaches provide several advantages:
  - More readable codes
  - Only one file
  - Independent from the hardware platform
  - Reasonable performance

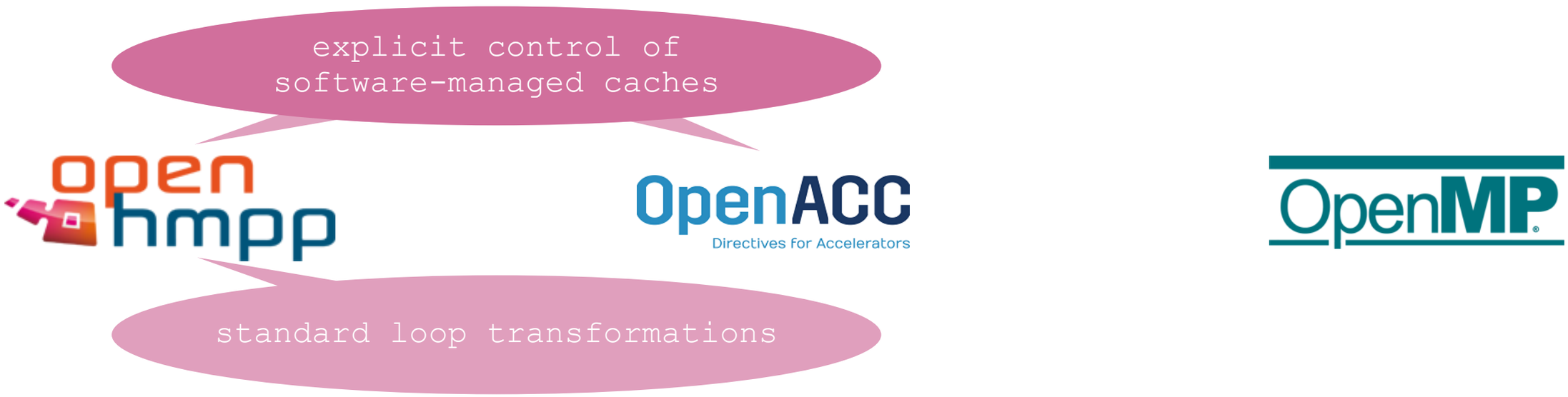
GPGPU with OpenHMPP

- Directive-based approaches provide several advantages:
  - More readable codes
  - Only one file
  - Independent from the hardware platform
  - Reasonable performance

explicit control of
software-managed caches

GPGPU with OpenHMPP

- Directive-based approaches provide several advantages:
  - More readable codes
  - Only one file
  - Independent from the hardware platform
  - Reasonable performance



GPU Programming Features with OpenHMPP

1	Threadification
2	Thread grouping
3	Minimization of CPU-GPU data transfers
4	Coalescing
5	Maximization of the usage of registers and shared memory
6	Divergency
7	Occupancy
8	Threads per block

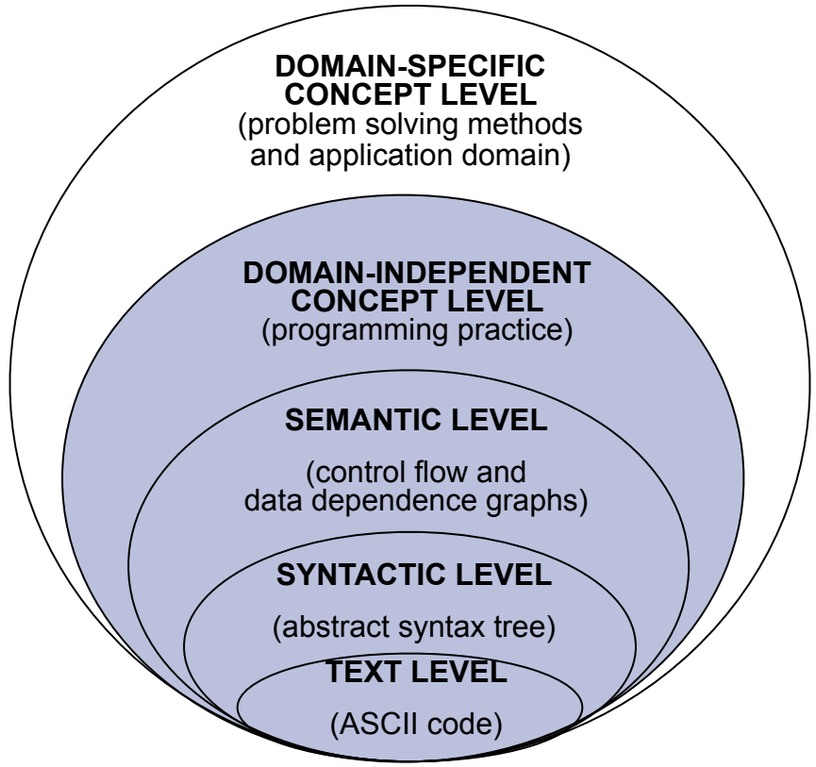
```
gridify
advancedLoad
delegatedStore
permute
unroll
fuse
tile
shared
```

J.M. Andión et al. Locality-Aware Automatic Parallelization for GPGPU with OpenHMPP Directives. HLPP 2014.

Outline

- Motivation: General Purpose Computation with GPUs
- GPGPU with CUDA & OpenHMPP
- The KIR: an IR for the Detection of Parallelism
- Locality-Aware Generation of Efficient GPGPU Code
- Case Studies: CONV3D & SGEMM
- Performance Evaluation
- Conclusions & Future Work

diKernel: Domain-Independent Computational Kernel



- Characterizes the computations carried out in a program without being affected by how they are coded

- Exposes multiple levels of parallelism

M. Arenaz et al. XARK: An Extensible Framework for Automatic Recognition of Computational Kernels. ACM Transactions on Programming Languages and Systems, 30(6), 2008.

Building the KIR

- Non-statement based, high-level, hierarchical IR

1. diKernel recognition on the DDG

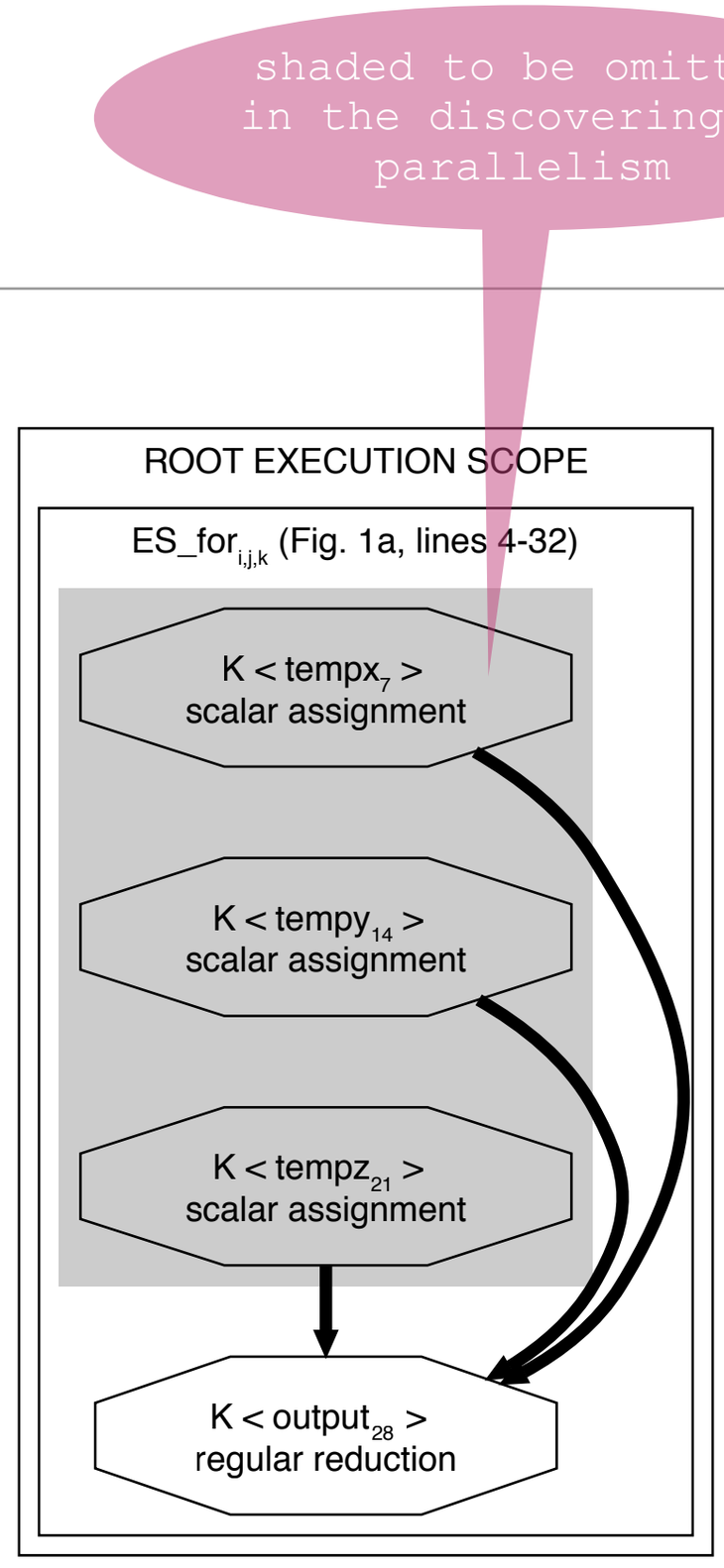
2. Identification of flow dependences

3. Hierarchy of execution scopes reflecting the computational stages & diKernel classification

J.M. Andión et al. A Novel Compiler Support for Automatic Parallelization on Multicore Systems. Parallel Computing, 39(9), 2013.

Example of KIR: CONV3D

```
1.int i, j, k, size_x, size_y, size_z;
2.float coefx,coefy,coefz,*input,*output;
3.
4.for (i = 0; i < size_x; i++) {
5. for (j = 0; j < size_y; j++) {
6.  for (k = 0; k < size_z; k++) {
7.   float tempx = input[i][j][k]+coefx*
8.     (
9.      input[i-1][j][k]+input[i+1][j][k]+
10.     input[i-2][j][k]+input[i+2][j][k]+
11.     input[i-3][j][k]+input[i+3][j][k]+
12.     input[i-4][j][k]+input[i+4][j][k]
13.    );
14.   float tempy = input[i][j][k]+coefy*
15.    (
16.     input[i][j-1][k]+input[i][j+1][k]+
17.     input[i][j-2][k]+input[i][j+2][k]+
18.     input[i][j-3][k]+input[i][j+3][k]+
19.     input[i][j-4][k]+input[i][j+4][k]
20.    );
21.   float tempz = input[i][j][k]+coefz*
22.    (
23.     input[i][j][k-1]+input[i][j][k+1]+
24.     input[i][j][k-2]+input[i][j][k+2]+
25.     input[i][j][k-3]+input[i][j][k+3]+
26.     input[i][j][k-4]+input[i][j][k+4]
27.    );
28.   output[i][j][k] =
29.    output[i][j][k]+tempx+tempy+tempz;
30.  }
31. }
32.}
```



Outline

- Motivation: General Purpose Computation with GPUs
- GPGPU with CUDA & OpenHMPP
- The KIR: an IR for the Detection of Parallelism
- **Locality-Aware Generation of Efficient GPGPU Code**
- Case Studies: CONV3D & SGEMM
- Performance Evaluation
- Conclusions & Future Work

GPU Programming Features addressed by our Automatic Technique

1	Threadification	✓
2	Thread grouping	✓
3	Minimization of CPU-GPU data transfers	
4	Coalescing	✓
5	Maximization of the usage of registers and shared memory	✓
6	Divergency	
7	Occupancy	
8	Threads per block	

Detection of coalesced accesses to the GPU global memory

CHRECS_x_k = [{0,+,1}][{0,+,1}]

```
1 // only for_i is threadified
2 for (i = 0; i <= N; i++) {
3    for (j = 0; j <= N; j++) {
4       ... x[i][j] ...
5    }
6 }
```

(a) Source code S1.

		T0 (i=0)	*T1* (i=1)	*T2* (i=2)
	j=0	$x[0][0]$	$x[1][0]$	$x[2][0]$
	j=1	$x[0][1]$	$x[1][1]$	$x[2][1]$
	j=2	$x[0][2]$	$x[1][2]$	$x[2][2]$
	...	...	...	...
chrecs	1^{st} *dim*	$\{0\}$	$\{1\}$	$\{2\}$
	2^{nd} *dim*	$\{0,+,1\}$	$\{0,+,1\}$	$\{0,+,1\}$

(b) Non-coalesced accesses.

```
1 // only for_j is threadified
2 for (j = 0; j <= N; j++) {
3    for (i = 0; i <= N; i++) {
4       ... x[i][j] ...
5    }
6 }
```

(c) Source code S2.

		T0 (j=0)	*T1* (j=1)	*T2* (j=2)
	i=0	$x[0][0]$	$x[0][1]$	$x[0][2]$
	i=1	$x[1][0]$	$x[1][1]$	$x[1][2]$
	i=2	$x[2][0]$	$x[2][1]$	$x[2][2]$
	...	...	...	...
chrecs	1^{st} *dim*	$\{0,+,1\}$	$\{0,+,1\}$	$\{0,+,1\}$
	2^{nd} *dim*	$\{0\}$	$\{1\}$	$\{2\}$

(d) Coalesced accesses.

Detection of coalesced accesses to the GPU global memory

CHRECS_x_k = [{0,+,1}][{0,+,1}]

```
1 // only for_i is threadified
2 for (i = 0; i <= N; i++) {
3     for (j = 0; j <= N; j++) {
4         ... x[i][j] ...
5     }
6 }
```

(a) Source code S1.

		T0 (i=0)	*T1* (i=1)	*T2* (i=2)
	j=0	$x[0][0]$	$x[1][0]$	$x[2][0]$
	j=1	$x[0][1]$	$x[1][1]$	$x[2][1]$
	j=2	$x[0][2]$	$x[1][2]$	$x[2][2]$
	...	...	...	...
chrecs	1^{st} *dim*	{0}	{1}	{2}
	2^{nd} *dim*	{0,+,1}	{0,+,1}	{0,+,1}

(b) Non-coalesced accesses.

```
1 // only for_j is threadified
2 for (j = 0; j <= N; j++) {
3     for (i = 0; i <= N; i++) {
4         ... x[i][j] ...
5     }
6 }
```

(c) Source code S2.

		T0 (j=0)	*T1* (j=1)	*T2* (j=2)
	i=0	$x[0][0]$	$x[0][1]$	$x[0][2]$
	i=1	$x[1][0]$	$x[1][1]$	$x[1][2]$
	i=2	$x[2][0]$	$x[2][1]$	$x[2][2]$
	...	...	...	...
chrecs	1^{st} *dim*	{0,+,1}	{0,+,1}	{0,+,1}
	2^{nd} *dim*	{0}	{1}	{2}

(d) Coalesced accesses.

the same

contiguous range

Locality-Aware Generation of Efficient GPGPU Code (and III)

2. Usage of registers to store reused data within a GPU thread

3. Usage of the GPU shared memory for data shared between the threads of a warp

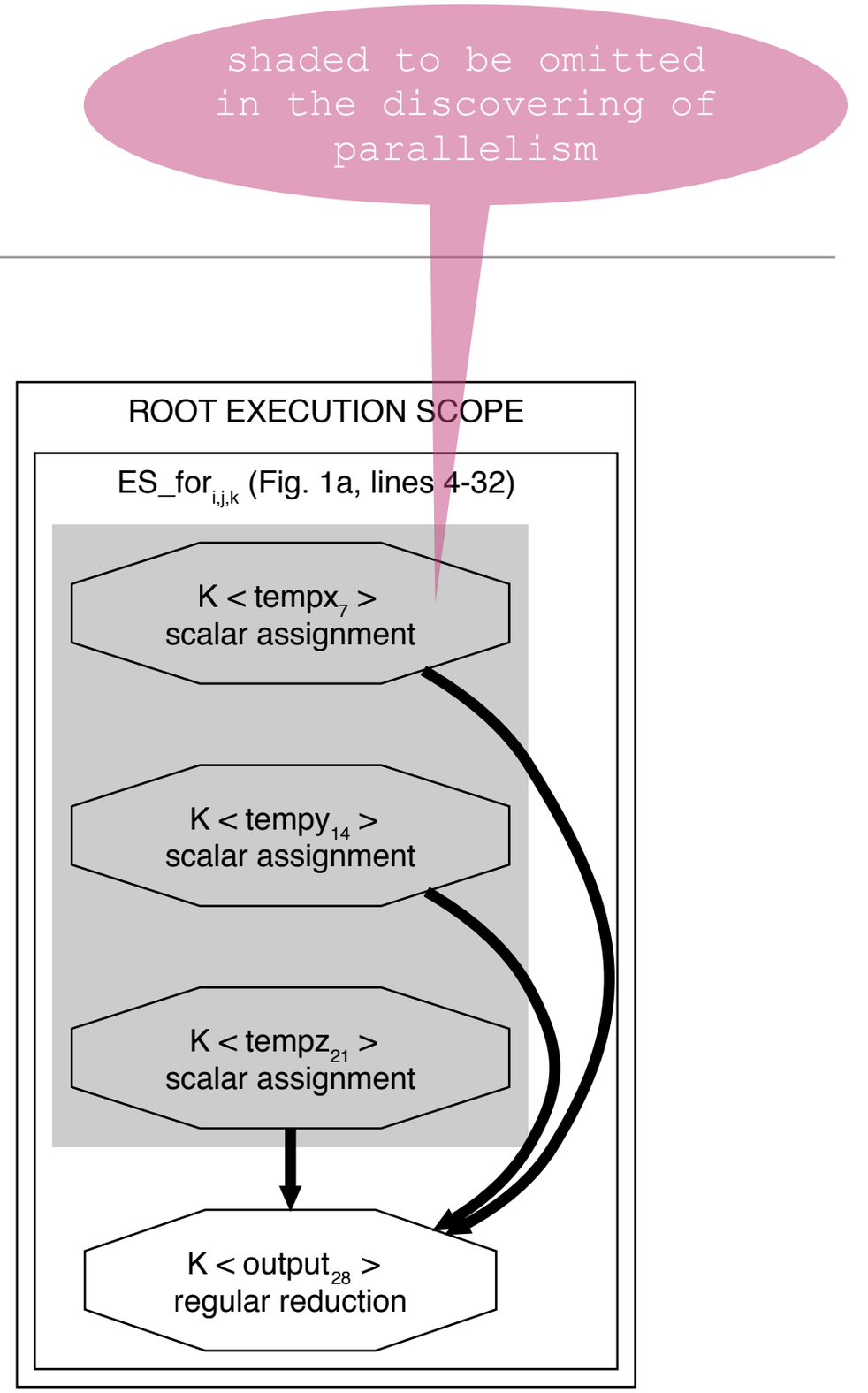
4. Increase the computational load of a GPU thread (loop tiling preserving coalescing)

Outline

- Motivation: General Purpose Computation with GPUs
- GPGPU with CUDA & OpenHMPP
- The KIR: an IR for the Detection of Parallelism
- Locality-Aware Generation of Efficient GPGPU Code
- **Case Studies: CONV3D & SGEMM**
- Performance Evaluation
- Conclusions & Future Work

Case Study: CONV3D (I)

```
1.int i, j, k, size_x, size_y, size_z;
2.float coefx,coefy,coefz,*input,*output;
3.
4.for (i = 0; i < size_x; i++) {
5. for (j = 0; j < size_y; j++) {
6.  for (k = 0; k < size_z; k++) {
7.   float tempx = input[i][j][k]+coefx*
8.     (
9.      input[i-1][j][k]+input[i+1][j][k]+
10.     input[i-2][j][k]+input[i+2][j][k]+
11.     input[i-3][j][k]+input[i+3][j][k]+
12.     input[i-4][j][k]+input[i+4][j][k]
13.    );
14.   float tempy = input[i][j][k]+coefy*
15.    (
16.     input[i][j-1][k]+input[i][j+1][k]+
17.     input[i][j-2][k]+input[i][j+2][k]+
18.     input[i][j-3][k]+input[i][j+3][k]+
19.     input[i][j-4][k]+input[i][j+4][k]
20.    );
21.   float tempz = input[i][j][k]+coefz*
22.    (
23.     input[i][j][k-1]+input[i][j][k+1]+
24.     input[i][j][k-2]+input[i][j][k+2]+
25.     input[i][j][k-3]+input[i][j][k+3]+
26.     input[i][j][k-4]+input[i][j][k+4]
27.    );
28.   output[i][j][k] =
29.    output[i][j][k]+tempx+tempy+tempz;
30.  }
31. }
32.}
```



Case Study: CONV3D (II)

- conv3d-cpu
- conv3d-hmpp1: Coalescing

```
1.int i, j, k, size_x, size_y, size_z;
2.float coefx,coefy,coefz,*input,*output;
3.
4.for (i = 0; i < size_x; i++) {
5. for (j = 0; j < size_y; j++) {
6.  for (k = 0; k < size_z; k++) {
7.   float tempx = input[i][j][k]+coefx*
…
```

$CHRECS\_input_1 = [\{0,+,1\}][\{0,+,1\}][\{0,+,1\}]$

- Default OpenHMPP policy

$CHRECS\_input_1^{T0} = [\{0\}][\{0\}][\{0,+,1\}]$

$CHRECS\_input_1^{T1} = [\{0\}][\{1\}][\{0,+,1\}]$

- Loop nest is permuted to *forj, fork, fori* (`permute` directive)

$CHRECS\_input_1^{T0} = [\{0,+,1\}][\{0\}][\{0\}]$

$CHRECS\_input_1^{T1} = [\{0,+,1\}][\{0\}][\{1\}]$

Case Study: CONV3D (III)

- conv3d-hmpp2: Registers

```
4.for (i = 0; i < size_x; i++) {
5. for (j = 0; j < size_y; j++) {
6.  for (k = 0; k < size_z; k++) {
7.   float tempx = input[i][j][k]+coefx*
8.     (
9.      input[i-1][j][k]+input[i+1][j][k]+
…
```

$CHRECS\_input_1 = [\{0,+,1\}][\{0,+,1\}][\{0,+,1\}]$

$CHRECS\_input_2 = [\{-1,+,1\}][\{0,+,1\}][\{0,+,1\}]$

$CHRECS\_input_1 = [\{1,+,1\}][\{0,+,1\}][\{0,+,1\}]$

$CHRECS\_input_1^{T0} = [\{0,+,1\}][\{0\}][\{0\}]$

$CHRECS\_input_2^{T0} = [\{-1,+,1\}][\{0\}][\{0\}]$

$CHRECS\_input_3^{T0} = [\{1,+,1\}][\{0\}][\{0\}]$

$$\cap \neq \emptyset$$

Case Study: CONV3D (and IV)

- conv3d-hmpp3: Shared memory

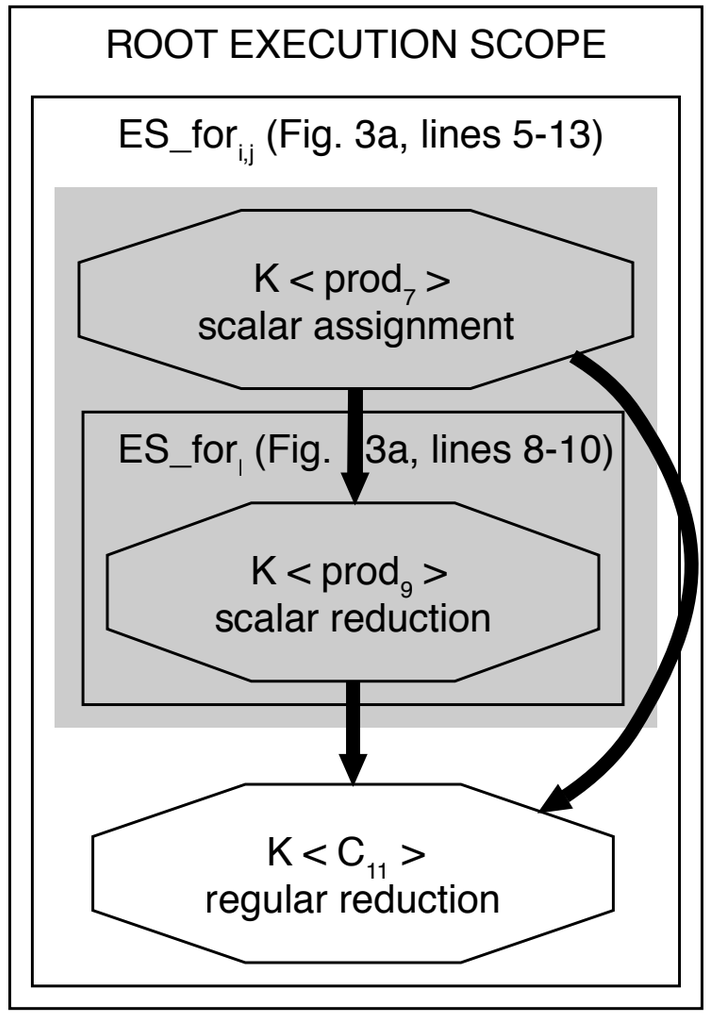
```
4.for (i = 0; i < size_x; i++) {
5. for (j = 0; j < size_y; j++) {
6.  for (k = 0; k < size_z; k++) {
…
21.  float tempz = input[i][j][k]+coefz*
22.    (
23.     input[i][j][k-1]+input[i][j][k+1]+
24.     input[i][j][k-2]+input[i][j][k+2]+
25.     input[i][j][k-3]+input[i][j][k+3]+
26.     input[i][j][k-4]+input[i][j][k+4]
27.    );
…
```

	T0			*T1*		
	$1^{st}\,dim$	$2^{nd}\,dim$	$3^{rd}\,dim$	$1^{st}\,dim$	$2^{nd}\,dim$	$3^{rd}\,dim$
$CHRECS\_input_{19}$	$\{0,+,1\}$	$\{0\}$	$\{0\}$	$\{0,+,1\}$	$\{0\}$	$\{1\}$
$CHRECS\_input_{20}$	$\{0,+,1\}$	$\{0\}$	$\{-1\}$	$\{0,+,1\}$	$\{0\}$	$\{0\}$
$CHRECS\_input_{21}$	$\{0,+,1\}$	$\{0\}$	$\{1\}$	$\{0,+,1\}$	$\{0\}$	$\{2\}$
$CHRECS\_input_{22}$	$\{0,+,1\}$	$\{0\}$	$\{-2\}$	$\{0,+,1\}$	$\{0\}$	$\{-1\}$
$CHRECS\_input_{23}$	$\{0,+,1\}$	$\{0\}$	$\{2\}$	$\{0,+,1\}$	$\{0\}$	$\{3\}$
$CHRECS\_input_{24}$	$\{0,+,1\}$	$\{0\}$	$\{-3\}$	$\{0,+,1\}$	$\{0\}$	$\{-2\}$
$CHRECS\_input_{25}$	$\{0,+,1\}$	$\{0\}$	$\{3\}$	$\{0,+,1\}$	$\{0\}$	$\{4\}$
$CHRECS\_input_{26}$	$\{0,+,1\}$	$\{0\}$	$\{-4\}$	$\{0,+,1\}$	$\{0\}$	$\{-3\}$
$CHRECS\_input_{27}$	$\{0,+,1\}$	$\{0\}$	$\{4\}$	$\{0,+,1\}$	$\{0\}$	$\{5\}$

`shared` clause of the `gridify` directive

Case Study: SGEMM (I)

```
1.int i, j, l, m, n, k;
2.float A[m][k], B[k][n], C[m][n];
3.float alpha, beta, prod;
4.
5.for (i = 0; i < m; i++) {
6.  for (j = 0; j < n; j++) {
7.    prod = 0;
8.    for (l = 0; l < k; l++) {
9.      prod += A[i][l] * B[l][j];
10.   }
11.   C[i][j] = alpha * prod + beta * C[i][j];
12. }
13.}
```



Case Study: SGEMM (II)

- sgemm-cpu
- sgemm-mkl: Intel MKL
- sgemm-hmpp1: Offloading (and check coalescing)

```
1.int i, j, l, m, n, k;
2.float A[m][k], B[k][n], C[m][n];
3.float alpha, beta, prod;
4.
5.for (i = 0; i < m; i++) {
6.  for (j = 0; j < n; j++) {
7.    prod = 0;
8.    for (l = 0; l < k; l++) {
9.      prod += A[i][l] * B[l][j];
10.   }
11.   C[i][j] = alpha * prod + beta * C[i][j];
12. }
13.}
```

	not instantiated		*T0*		*T1*	
	1^{st} *dim*	2^{nd} *dim*	1^{st} *dim*	2^{nd} *dim*	1^{st} *dim*	2^{nd} *dim*
CHRECS_A	$\{0,+,1\}$	$\{0,+,1\}$	$\{0\}$	$\{0,+,1\}$	$\{0\}$	$\{0,+,1\}$
CHRECS_B	$\{0,+,1\}$	$\{0,+,1\}$	$\{0,+,1\}$	$\{0\}$	$\{0,+,1\}$	$\{1\}$
CHRECS_C	$\{0,+,1\}$	$\{0,+,1\}$	$\{0\}$	$\{0\}$	$\{0\}$	$\{1\}$

Case Study: SGEMM (and III)

- sgemm-hmpp2: Tiling preserving coalescing

```
1.int i, j, l, m, n, k;
2.float A[m][k], B[k][n], C[m][n];
3.float alpha, beta, prod;
4.
5.for (i = 0; i < m; i++) {
6.  for (j = 0; j < n; j++) {
7.    prod = 0;
8.    for (l = 0; l < k; l++) {
9.      prod += A[i][l] * B[l][j];
10.   }
11.   C[i][j] = alpha * prod + beta * C[i][j];
12. }
13.}
```

Algorithm 4 Increase the computational load of a GPU thread

1: **procedure** INCREASELOAD
Input: access $x_k[i_{k,1}][i_{k,2}]\dots[i_{k,n}]$ to an n-dimensional array x stored in row-major order
Input: loop nest $L = L_1, L_2, \dots, L_l$ where both L_1, L_2 are threadified
Input: amount of data Δ to be processed by a GPU thread
2: increment the step of the outer loop L_1 to Δ
3: **for each** scalar variable s in L **do**
4: promote s to an array $s[\Delta]$
5: transform reads and writes to s into loops of Δ iterations
6: **end for**
7: **end procedure**

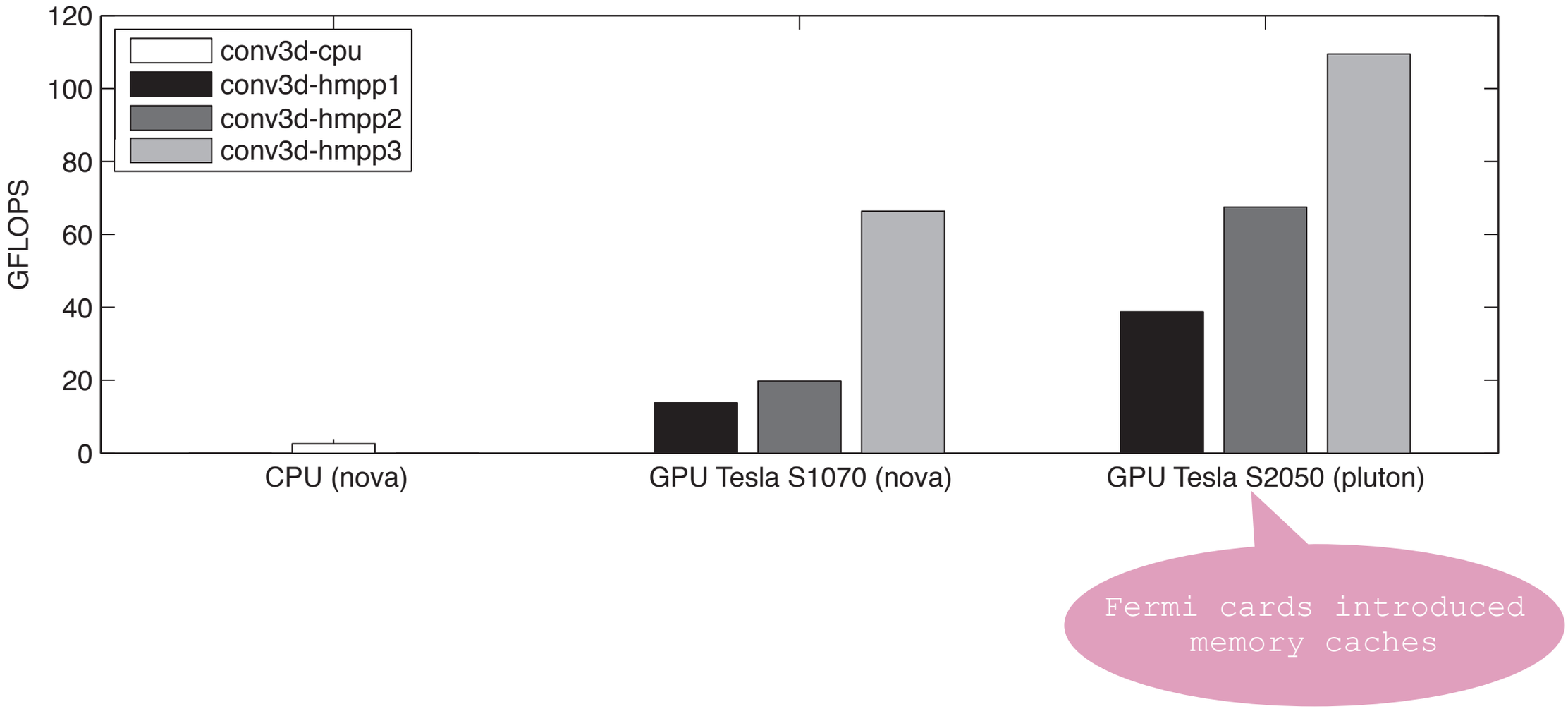
- sgemm-hmpp3: Let the compiler use the registers (`unroll`)
- sgemm-hmpp4: Use the shared memory for *B*
- sgemm-cublas: NVIDIA CUBLAS library

Outline

- Motivation: General Purpose Computation with GPUs
- GPGPU with CUDA & OpenHMPP
- The KIR: an IR for the Detection of Parallelism
- Locality-Aware Generation of Efficient GPGPU Code
- Case Studies: CONV3D & SGEMM
- **Performance Evaluation**
- Conclusions & Future Work

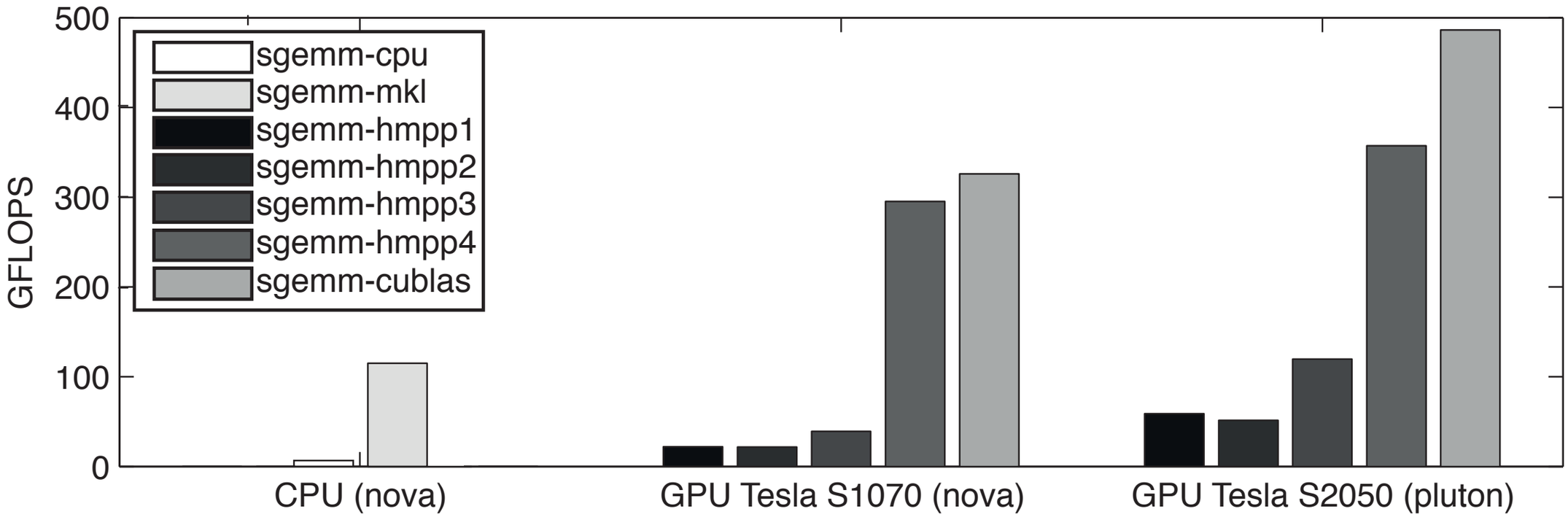
Performance Evaluation: CONV3D

$size_X$, $size_Y$ and $size_Z$ in 128, 256, 384, 512, 640 and 768



Performance Evaluation: SGEMM

m, n and k in 128, 256, 384, 512, 640, 768, 896, 1024, 1152, 1280, 1408, 1536, 1664, 1792, 1920, 2048, 4096, 6144 and 8192



the biggest improvement factor is the usage of the GPU shared memory

Outline

- Motivation: General Purpose Computation with GPUs
- GPGPU with CUDA & OpenHMPP
- The KIR: an IR for the Detection of Parallelism
- Locality-Aware Generation of Efficient GPGPU Code
- Case Studies: CONV3D & SGEMM
- Performance Evaluation
- **Conclusions & Future Work**

Conclusions

KIR-based locality-aware automatic parallelization technique that targets GPU-based heterogeneous systems

- exploit data locality in the complex GPU memory hierarchy
- two representative case studies: CONV3D & SGEMM
- chains of recurrences model accesses to n-dimensional arrays
- OpenHMPP directives enabled a great understandability and portability of the generated GPU code

Future Work

- New automatic partitioning algorithm of the KIR to handle the interactions between computations in full-scale applications

- Auto-tuning approaches to select the best performance on a given hardware architecture

- Test with larger benchmark suite and on other manycore accelerators

Locality-Aware Automatic Parallelization for GPGPU with OpenHMPP Directives

José M. Andión, Manuel Arenaz, François Bodin, Gabriel Rodríguez and Juan Touriño

7th International Symposium on High-Level Parallel Programming and Applications (HLPP 2014)
July 3-4, 2014 — Amsterdam, Netherlands

COMPUTER ARCHITECTURE GROUP
UNIVERSITY OF A CORUÑA
GAC.UDC.ES
UNIVERSIDADE DA CORUÑA