FCUDA: Enabling Efficient Compilation of CUDA Kernels onto FPGAs

Andrew Canis

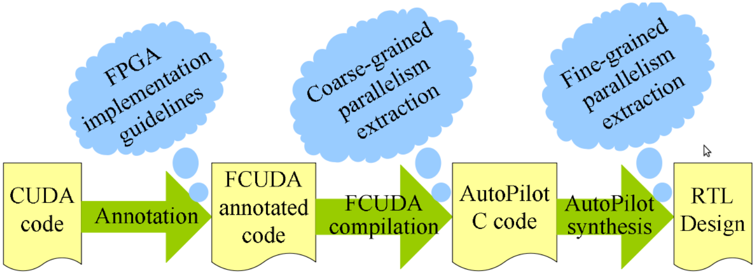
October 13, 2009

Overview

- Presenting:
  - Alex Papakonstantinou, Karthik Gururaj, John Stratton, Jason Cong, Deming Chen, Wen-mei Hwu. **FCUDA: Enabling Efficient Compilation of CUDA Kernels onto FPGAs**. Symposium on Application Specific Processors, July, 2009. (Best Paper Award)
- Overview
- Comparison: FPGA vs GPU vs CPU
- CUDA
- FCUDA
- AutoPilot
- Results

FCUDA: Overview

- CUDA
  - popular language for programming Nvidia GPUs
  - describes coarse-grained parallelism
- FCUDA: convert CUDA into annotated C code
- Use high level synthesis (AutoPilot) to convert C into RTL
- Synthesize RTL for an FPGA


FPGA implementation guidelines
Coarse-grained parallelism extraction
Fine-grained parallelism extraction
CUDA code
Annotation
FCUDA annotated code
FCUDA compilation
AutoPilot C code
AutoPilot synthesis
RTL Design
Fig. 1. CUDA-to-FPGA Flow

Why FPGAs?

- FPGAs have higher computational density per Watt than GPUs and CPUs:
  - 32-bit integer arithmetic: 6X higher
  - 32-bit floating point arithmetic: 2X higher

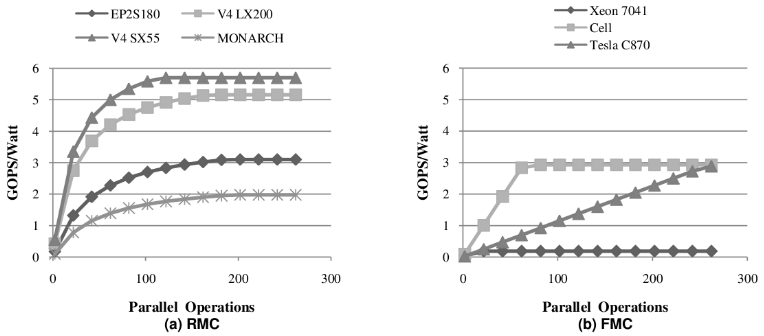
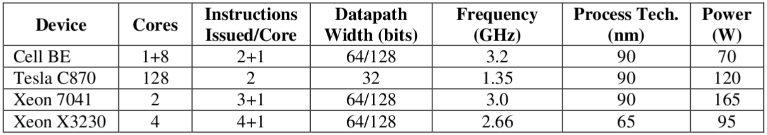


Figure 7. SPFP CDW (90 nm)

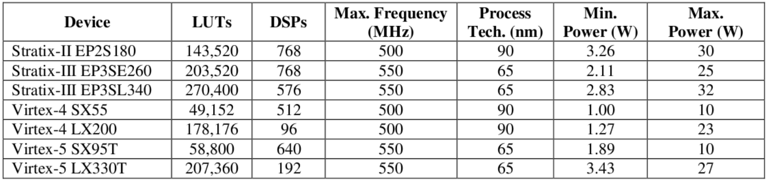
Power: FPGA vs GPU vs CPU

Table 2. FMC Device Features



Device	Cores	Instructions Issued/Core	Datapath Width (bits)	Frequency (GHz)	Process Tech. (nm)	Power (W)
Cell BE	1+8	2+1	64/128	3.2	90	70
Tesla C870	128	2	32	1.35	90	120
Xeon 7041	2	3+1	64/128	3.0	90	165
Xeon X3230	4	4+1	64/128	2.66	65	95

Table 3. FPGA Device Features

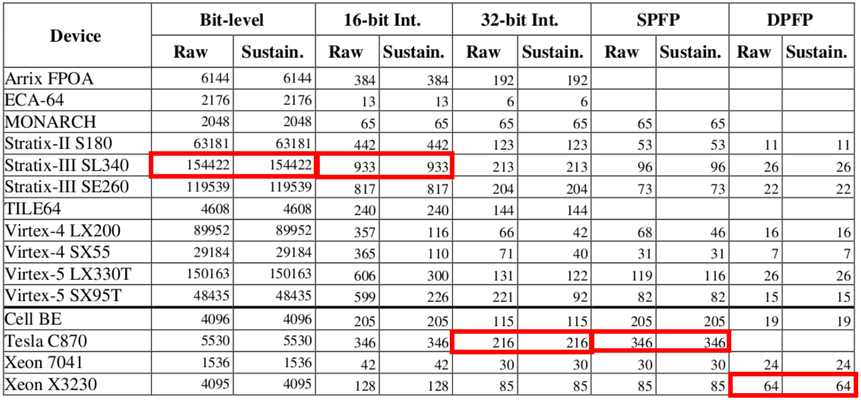


Device	LUTs	DSPs	Max. Frequency (MHz)	Process Tech. (nm)	Min. Power (W)	Max. Power (W)
Stratix-II EP2S180	143,520	768	500	90	3.26	30
Stratix-III EP3SE260	203,520	768	550	65	2.11	25
Stratix-III EP3SL340	270,400	576	550	65	2.83	32
Virtex-4 SX55	49,152	512	500	90	1.00	10
Virtex-4 LX200	178,176	96	500	90	1.27	23
Virtex-5 SX95T	58,800	640	550	65	1.89	10
Virtex-5 LX330T	207,360	192	550	65	3.43	27

- J. Williams et. al. "Computational density of fixed and reconfigurable multi-core devices for application acceleration", RSSI, 2008.

Computational Density: FPGA vs GPU vs CPU

Table 5. CD (in billions of operations per second or GOPs)



Device	Bit-level		16-bit Int.		32-bit Int.		SPFP		DPFP	
	Raw	Sustain.	Raw	Sustain.	Raw	Sustain.	Raw	Sustain.	Raw	Sustain.
Arrix FPOA	6144	6144	384	384	192	192				
ECA-64	2176	2176	13	13	6	6				
MONARCH	2048	2048	65	65	65	65	65	65		
Stratix-II S180	63181	63181	442	442	123	123	53	53	11	11
Stratix-III SL340	154422	154422	933	933	213	213	96	96	26	26
Stratix-III SE260	119539	119539	817	817	204	204	73	73	22	22
TILE64	4608	4608	240	240	144	144				
Virtex-4 LX200	89952	89952	357	116	66	42	68	46	16	16
Virtex-4 SX55	29184	29184	365	110	71	40	31	31	7	7
Virtex-5 LX330T	150163	150163	606	300	131	122	119	116	26	26
Virtex-5 SX95T	48435	48435	599	226	221	92	82	82	15	15
Cell BE	4096	4096	205	205	115	115	205	205	19	19
Tesla C870	5530	5530	346	346	216	216	346	346		
Xeon 7041	1536	1536	42	42	30	30	30	30	24	24
Xeon X3230	4095	4095	128	128	85	85	85	85	64	64

- J. Williams et. al. "Computational density of fixed and reconfigurable multi-core devices for application acceleration", RSSI, 2008.

CUDA: Compute Unified Device Architecture

- CUDA is a C API and compiler infrastructure for Nvidia GPUs
- GPU is a SIMD architecture

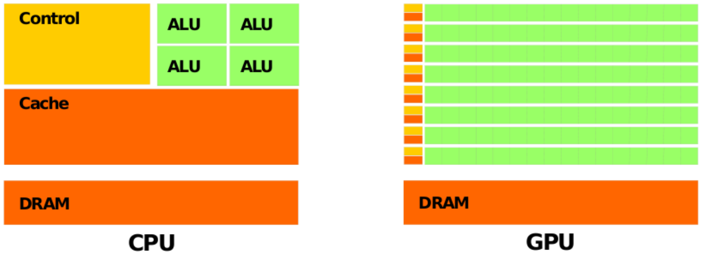
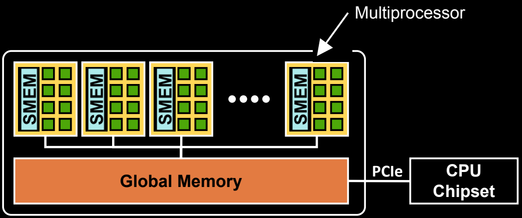
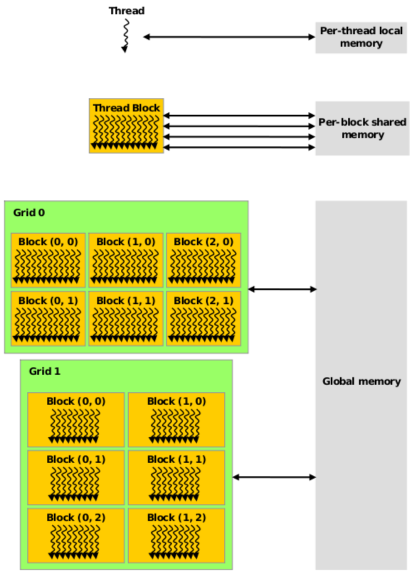


Figure 1-2. The GPU Devotes More Transistors to Data Processing

Simple Hardware View



GPU Memory Hierarchy



CUDA Example

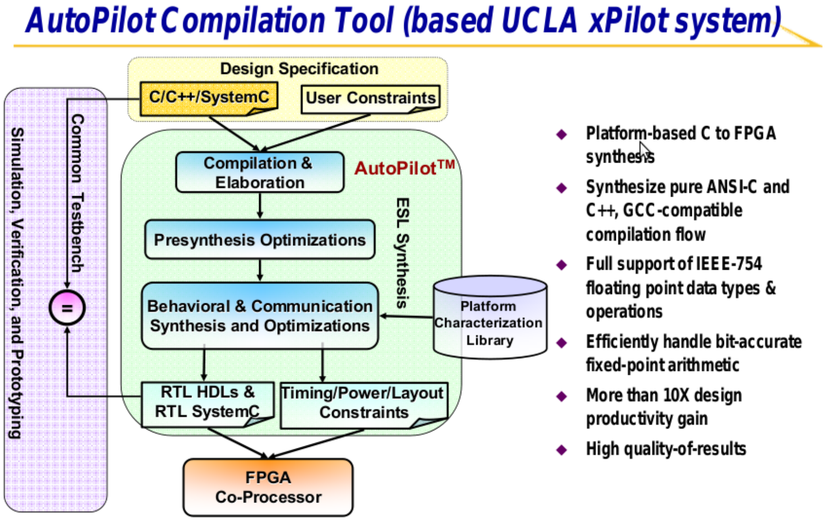
C Code:

```
int main() {
    ...
    for (int i = 0; i < N; i++) {
        C[i] = A[i] + B[i];
    }
}
```

CUDA:

```
int main() {
    ...
    VecAdd<<<1, N>>>(A, B, C);
}
__global__ void VecAdd(float* A, float* B, float* C) {
  int i = threadIdx.x;
   C[i] = A[i] + B[i];
}
```

AutoPilot



AutoPilot Compilation Tool (based UCLA xPilot system)

Design Specification
C/C++/SystemC
User Constraints
Compilation & Elaboration
AutoPilot™
Presynthesis Optimizations
ESL Synthesis
Behavioral & Communication Synthesis and Optimizations
Platform Characterization Library
RTL HDLs & RTL SystemC
Timing/Power/Layout Constraints
FPGA Co-Processor
Simulation, Verification, and Prototyping
Common Testbench
=

- Platform-based C to FPGA synthesis
- Synthesize pure ANSI-C and C++, GCC-compatible compilation flow
- Full support of IEEE-754 floating point data types & operations
- Efficiently handle bit-accurate fixed-point arithmetic
- More than 10X design productivity gain
- High quality-of-results

AutoPilot

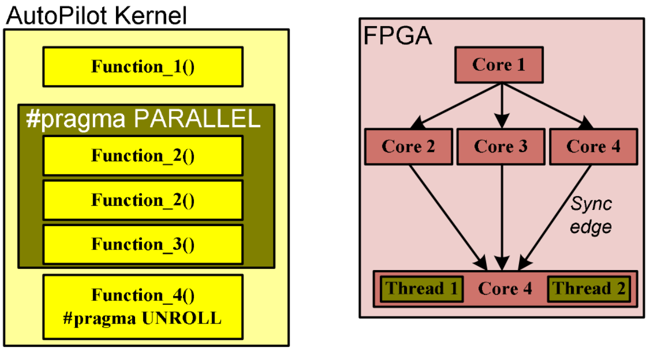


Fig. 3. AutoPilot C Programming Model

FCUDA

- Map thread-blocks onto parallel FPGA cores
  - Minimize inter-core communication to avoid synchronization
- FCUDA pragmas:
  - COMPUTE: computation task of kernel
  - TRANSFER: data communication task to off-chip DRAM
  - SYNC: synchronization scheme
    - simple DMA: serialize data communication and computation
    - ping-pong DMA: double BRAMs to allow concurrent data communication and computation

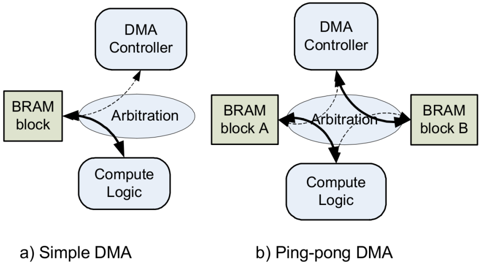


Fig. 4. Scheduling Schemes

Coulombic Potential (CP) CUDA kernel

```



__global__ void cenergy(int numatoms, int gridspacing, int *
    energygrid) {




    ...
    int energyval = 0;
    for (atomid=0; atomid<numatoms; atomid++) {
        ...
        energyval += atominfo[atomid].w * r_1;
    }


    energygrid[outaddr] += energyval;

}
```

FCUDA-annotated CP kernel

```
#pragma FCUDA GRID x_dim=2 y_dim=1 begin name="cp_grid"
#pragma FCUDA BLOCKS start_x=0 end_x=128 start_y=0 end_y=128
#pragma FCUDA SYNC type=simple
__global__ void cenergy(int numatoms, int gridspacing, int *
        energygrid) {




    ...
    int energyval = 0;
    for (atomid=0; atomid<numatoms; atomid++) {
        ...
        energyval += atominfo[atomid].w * r_1;
    }


    energygrid[outaddr] += energyval;

}
#pragma FCUDA GRID end name="cp_grid"
```

FCUDA-annotated CP kernel

```
#pragma FCUDA GRID x_dim=2 y_dim=1 begin name="cp_grid"
#pragma FCUDA BLOCKS start_x=0 end_x=128 start_y=0 end_y=128
#pragma FCUDA SYNC type=simple
__global__ void cenergy(int numatoms, int gridspacing, int *
    energygrid) {



#pragma FCUDA COMPUTE cores=2 begin name="cp_block"
    ...
    int energyval = 0;
    for (atomid=0; atomid<numatoms; atomid++) {
        ...
        energyval += atominfo[atomid].w * r_1;
    }
#pragma FCUDA COMPUTE end name="cp_block"

    energygrid[outaddr] += energyval;

}
#pragma FCUDA GRID end name="cp_grid"
```

FCUDA-annotated CP kernel

```
#pragma FCUDA GRID x_dim=2 y_dim=1 begin name="cp_grid"
#pragma FCUDA BLOCKS start_x=0 end_x=128 start_y=0 end_y=128
#pragma FCUDA SYNC type=simple
 __global__ void cenergy(int numatoms, int gridspacing, int *
      energygrid) {
#pragma FCUDA TRANSFER cores=1 type=burst begin name="fetch"
#pragma FCUDA DATA type=load from=atominfo start=0 end=
     MAXATOMS
#pragma FCUDA TRANSFER end name="fetch"
#pragma FCUDA COMPUTE cores=2 begin name="cp_block"
     ...
     int energyval = 0;
     for (atomid=0; atomid<numatoms; atomid++) {
         ...
         energyval += atominfo[atomid].w * r_1;
     }
#pragma FCUDA COMPUTE end name="cp_block"
#pragma FCUDA TRANSFER cores=1 type=burst begin name="write"
     energygrid[outaddr] += energyval;
#pragma FCUDA TRANSFER end name="write"
}
#pragma FCUDA GRID end name="cp_grid"
```

FCUDA compiler passes

- FCUDA based on Cetus source-to-source compiler
- Front-end transformations:
  - Convert implicit thread execution into an explicit loop
  - Vectorize local thread variables
- Back-end transformation:
  - Split tasks into separate AutoPilot functions
  - Wrap parallel function calls with AUTOPILOT REGION PARALLEL pragmas
  - Create DMA burst transfers between off-chip DRAM and on-chip BRAM arrays

FCUDA front-end processed CP compute task

```
#pragma FCUDA COMPUTE cores=2 begin name="cp_block"



              ...
              int energyval = 0;
              for (atomid=0; atomid<numatoms; atomid++) {
                   ...
                   energyval += atominfo[atomid].w * r_1;
              }
         }
    }
#pragma FCUDA COMPUTE end name="cp_block"
```

FCUDA front-end processed CP compute task

```
#pragma FCUDA COMPUTE cores=2 begin name="cp_block"
    int energyval[];
    for(tIdx.y=0;tIdx.y<blockDim;tIdx.y++) // thread loop
        for(tIdx.x=0;tIdx.x<blockDim;tIdx.x++) {
            ...
            energyval[tIdx]=0.0f;
            for (atomid=0; atomid<numatoms; atomid++) {
                ...
                energyval[tIdx] += atominfo[atomid].w * r_1;
            }
        }
    }
#pragma FCUDA COMPUTE end name="cp_block"
```

FCUDA-generated AutoPilot code

```
void cp_block(blockIdx, blockDim, energyval[], atominfo[])  {
   for(tIdx.y=0;tIdx.y<blockDim;tIdx.y++)   // thread loop
    for(tIdx.x=0;tIdx.x<blockDim;tIdx.x++) {
…
       }
     }
}


void fetch (blockIdx, blockDim, atominfo, c1_atominfo[], c2_atominfo[]) {
  …
}

void write (blockIdx, blockDim, c1_energyval[],c2_energyval[], energygrid) {
  …
}
```

Task functions

```
void cenergy(int numatoms, int gridspacing, int * energygrid,
                      dim3 blockDim, dim3 gridDim) {
dim3 tIdx, bIdx;  // thread and thread-block index structures
int c1_atominfo[], c2_atominfo[];
int c1_energyval[], c2_energyval[];
for(bIdx.y=0;bIdx.y<blockDim;bIdx.y++)   // block loop
 for(bIdx.x=0;bIdx.x<blockDim;bIdx.x+=2) {

   fetch(bIdx, blockDim, atominfo, c1_atominfo[], c2_atominfo[]) ;


#pragma AUTOPILOT REGION begin name="R1"
#pragma AUTOPILOT PARALLEL
   cp_block(blockIdx, blockDim, c1_energyval[], c1_atominfo[]) ;
   cp_block(blockIdx+1, blockDim, c2_energyval[], c2_atominfo[]) ;
#pragma AUTOPILOT REGION end name="R1"

   write (blockIdx, blockDim, c1_energyval[],c2_energyval[], energygrid);
  }
}
```

Parent function

Results: CUDA kernels

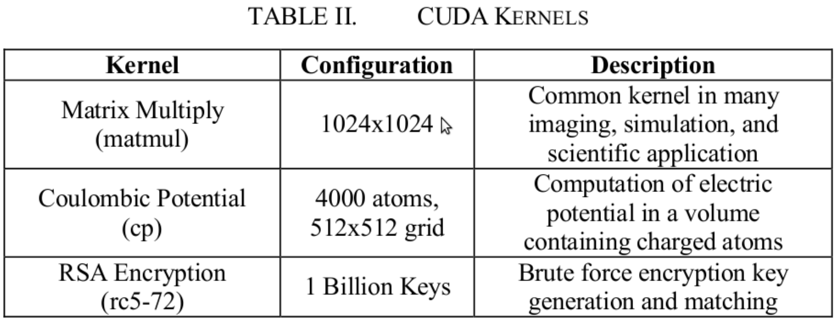
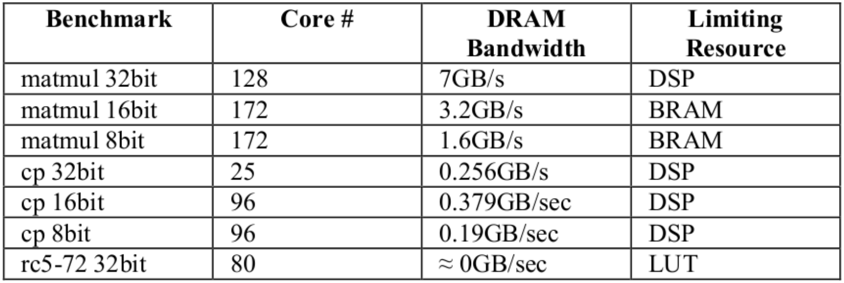


TABLE II. CUDA KERNELS

Kernel	Configuration	Description
Matrix Multiply (matmul)	1024x1024	Common kernel in many imaging, simulation, and scientific application
Coulombic Potential (cp)	4000 atoms, 512x512 grid	Computation of electric potential in a volume containing charged atoms
RSA Encryption (rc5-72)	1 Billion Keys	Brute force encryption key generation and matching

Results: FPGA kernel characteristics

TABLE III. KERNEL IMPLEMENTATION CHARACTERISTICS



Benchmark	**Core #**	**DRAM Bandwidth**	**Limiting Resource**
matmul 32bit	128	7GB/s	DSP
matmul 16bit	172	3.2GB/s	BRAM
matmul 8bit	172	1.6GB/s	BRAM
cp 32bit	25	0.256GB/s	DSP
cp 16bit	96	0.379GB/sec	DSP
cp 8bit	96	0.19GB/sec	DSP
rc5-72 32bit	80	≈ 0GB/sec	LUT

Results: Devices

- FPGA: Virtex5 XC5VFX200T device (65nm)
  - Over 100K LUTs
  - 2MB of on-chip BlockRAM memory
  - 384 DSP units
  - No specs given for off-chip DRAM
  - Max clock: 550Mhz
- GPU: Nvidia GeForce 8800 GTX (90nm)
  - 16 Streaming Multiprocessors (SM) and 128 cores.
  - Register File: 32KB/SM = 512KB total
  - Shared (on-chip) memory: 16KB/SM = 256KB total
  - Global (off-chip) memory 768MB, 86.4GB/sec bandwidth
  - Core Clock: 575Mhz,

Results: FPGA vs GPU Speedup

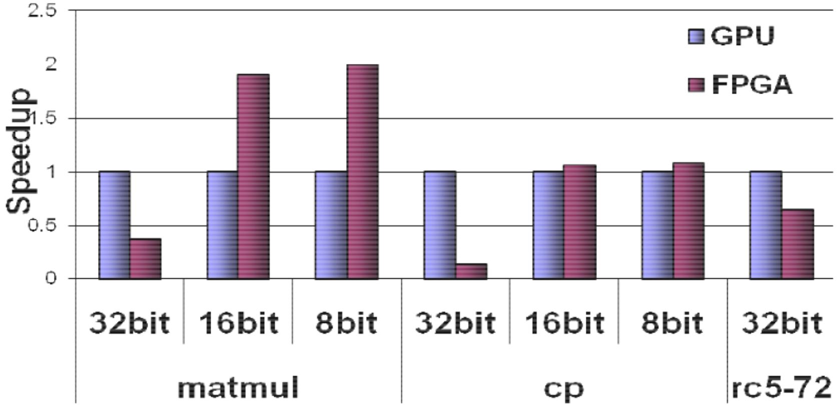
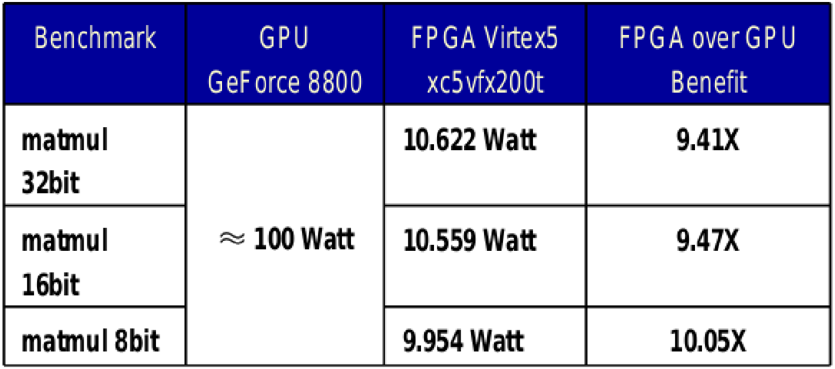


Fig. 10. GPU – FPGA Performance comparison

Results: FPGA vs GPU Power



Benchmark	GPU GeForce 8800	FPGA Virtex5 xc5vfx200t	FPGA over GPU Benefit
matmul 32bit	$\approx$ 100 Watt	10.622 Watt	9.41X
matmul 16bit		10.559 Watt	9.47X
matmul 8bit		9.954 Watt	10.05X

Conclusions

- CUDA to FPGA flow is possible with decent performance
- No comparison to more recent GPUs (GTX280)
- Future work: use OpenCL instead of CUDA
  - OpenCL supports for both ATI and Nvidia GPUs

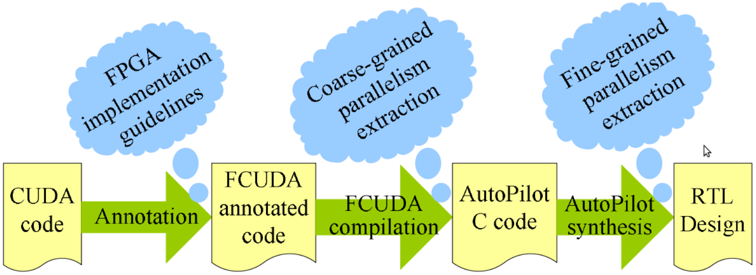
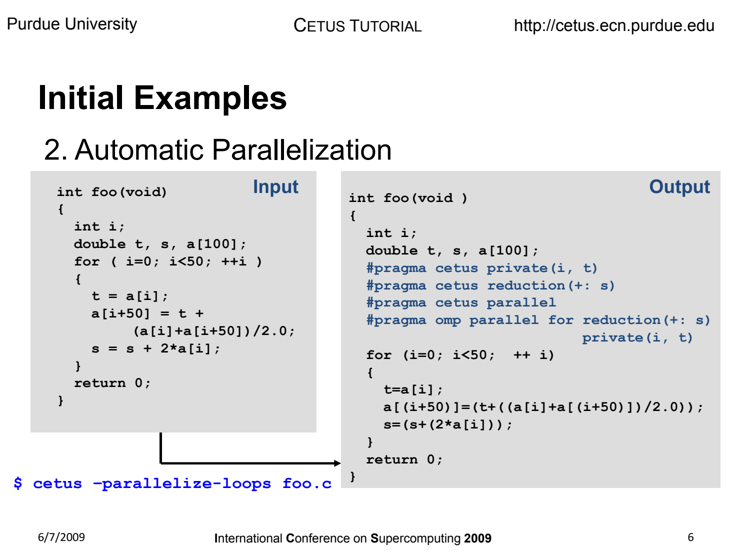

FPGA implementation guidelines
Coarse-grained parallelism extraction
Fine-grained parallelism extraction
CUDA code
Annotation
FCUDA annotated code
FCUDA compilation
AutoPilot C code
AutoPilot synthesis
RTL Design

Fig. 1. CUDA-to-FPGA Flow

Cetus: Source-to-source compiler

- Cetus is a source-to-source C compiler written in Java
- Maintained by Purdue University
- Designed for automatic parallelization
  - Data dependence analysis
  - Control flow graph generation
- Available at cetus.ecn.purdue.edu
  - Modified Artistic License

Cetus: Examples



Purdue University CETUS TUTORIAL http://cetus.ecn.purdue.edu

Initial Examples

2. Automatic Parallelization

Input

```
int foo(void)
{
  int i;
  double t, s, a[100];
  for ( i=0; i<50; ++i )
  {
    t = a[i];
    a[i+50] = t +
         (a[i]+a[i+50])/2.0;
    s = s + 2*a[i];
  }
  return 0;
}
```

```
$ cetus -parallelize-loops foo.c
```

Output

```
int foo(void )
{
  int i;
  double t, s, a[100];
  #pragma cetus private(i, t)
  #pragma cetus reduction(+: s)
  #pragma cetus parallel
  #pragma omp parallel for reduction(+: s)
                           private(i, t)
  for (i=0; i<50;  ++ i)
  {
    t=a[i];
    a[(i+50)]=(t+((a[i]+a[(i+50)])/2.0));
    s=(s+(2*a[i]));
  }
  return 0;
}
```

6/7/2009 International Conference on Supercomputing 2009 6