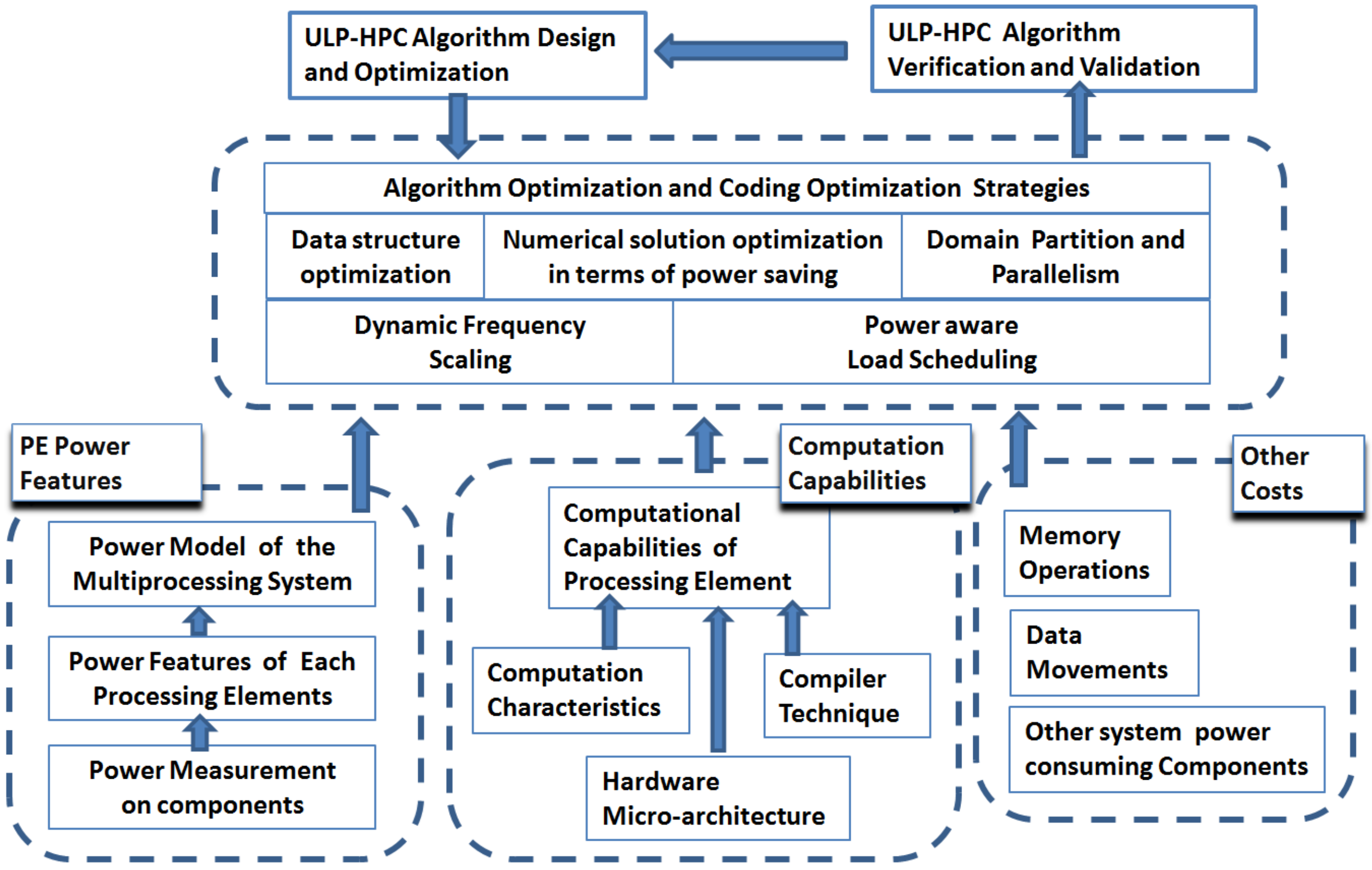
Optimization on the Power Efficiency of GPU and Multicore Processing Element for SIMD Computing

Da-Qi Ren and Reiji Suda
Department of Computer Science
THE UNIVERSITY OF TOKYO

Energy aware SIMD/SPMD program design framework



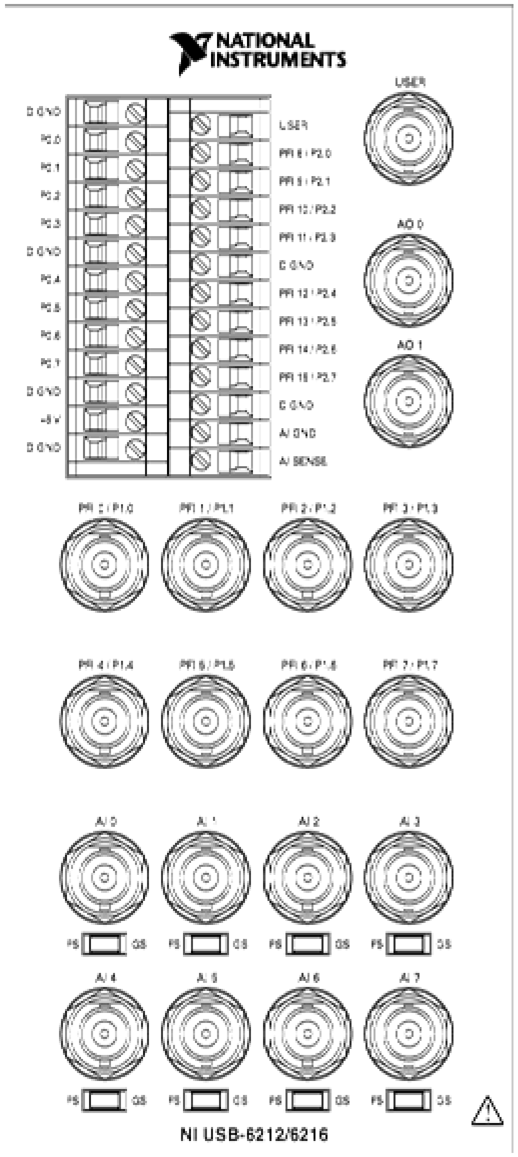
ULP-HPC Algorithm Design and Optimization
ULP-HPC Algorithm Verification and Validation
Algorithm Optimization and Coding Optimization Strategies
Data structure optimization
Numerical solution optimization in terms of power saving
Domain Partition and Parallelism
Dynamic Frequency Scaling
Power aware Load Scheduling
PE Power Features
Power Model of the Multiprocessing System
Power Features of Each Processing Elements
Power Measurement on components
Computation Capabilities
Computational Capabilities of Processing Element
Computation Characteristics
Compiler Technique
Hardware Micro-architecture
Other Costs
Memory Operations
Data Movements
Other system power consuming Components

GOAL: Importing hardware power parameters to software algorithm design, for improving the software energy efficiency.

1. **CUDA Processing Element (PE) Power Feature Determination:** measurements (Flops/watt) ;
2. **PE Computation Capability:** micro-architecture, language, compiler and characters of the computation;
3. **Algorithm and Code Optimization Strategies:** computer resources and power consumption.
4. **Verification and Validation:** incremental procedure.

Measurement instruments and environment setup

- National Instruments USB-6216 BNC data acquisition

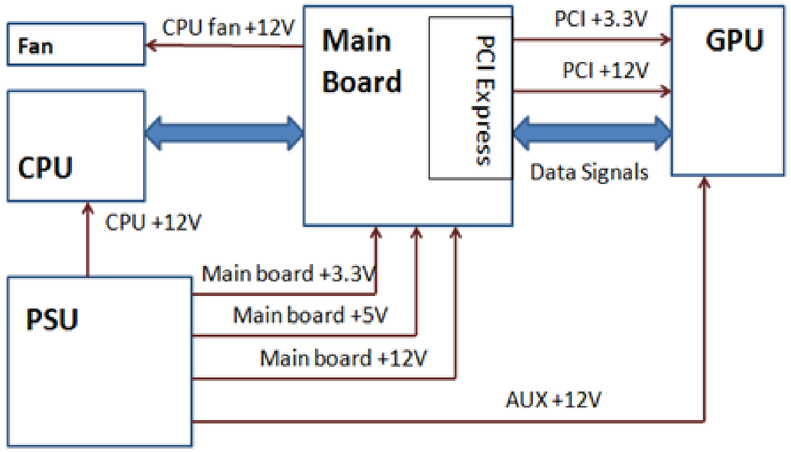


Fluke i30s / i310s current probes

Yokogawa 700925 voltage probe

- The room was air-conditioned in 23◦C. LabView 8.5 as oscilloscopes and analyzer for result data analysis.

- Real time voltage and current from measurement readings; their product is the instant power at each sampling point.



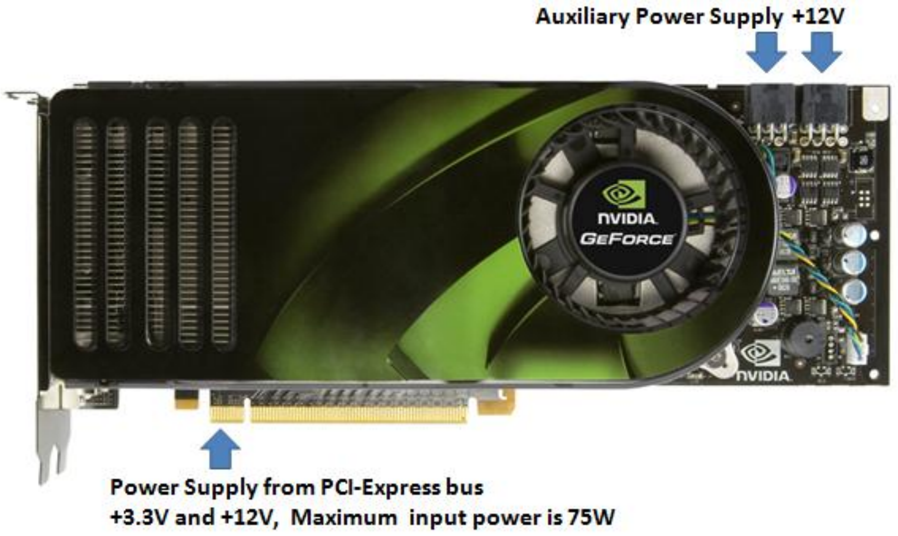
Power Measurement of GPU

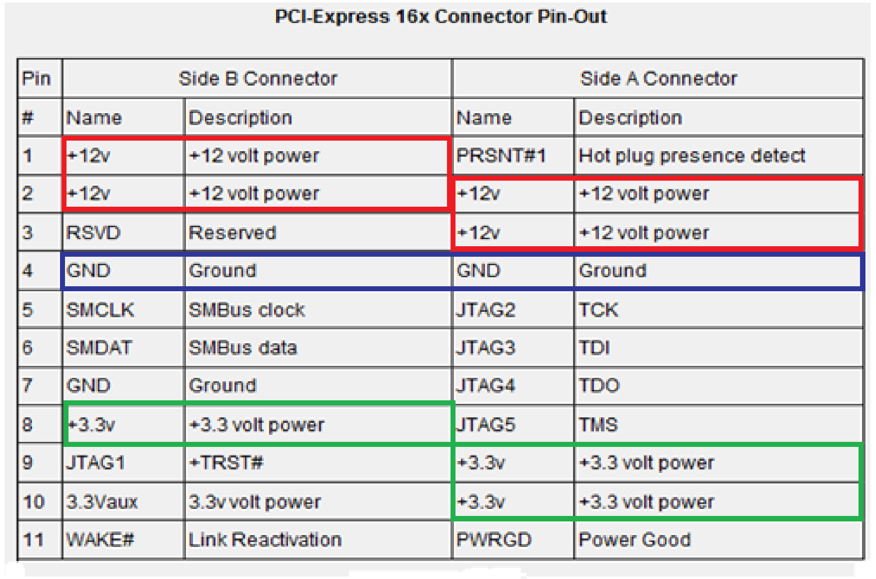
A GPU card is plugged in a PCI-Express slot on main board, it is mainly powered by

- ☐ +12V power from PCI-Express pins
- ☐ +3.3V power from PCI-Express pins
- ☐ An additional +12V power directly from PSU (because sometimes the PCI-E power may not be enough to support the GPU's high performance computation).

☐ Auxiliary power is measured through the auxiliary power line;

☐ A riser card to connect in between the PCI-Express slot and the GPU plug, in order to measure the pins.


Auxiliary Power Supply +12V
Power Supply from PCI-Express bus
+3.3V and +12V, Maximum input power is 75W

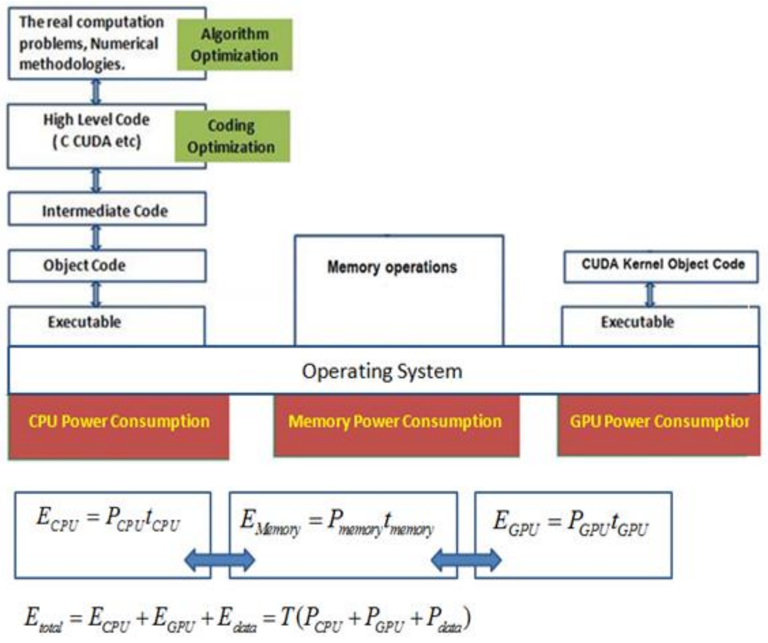


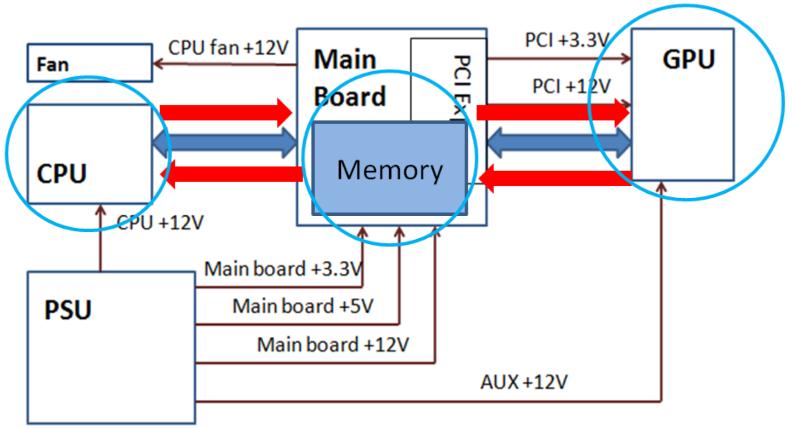
PCI-Express 16x Connector Pin-Out

Pin	Side B Connector		Side A Connector	
#	Name	Description	Name	Description
1	+12v	+12 volt power	PRSNT#1	Hot plug presence detect
2	+12v	+12 volt power	+12v	+12 volt power
3	RSVD	Reserved	+12v	+12 volt power
4	GND	Ground	GND	Ground
5	SMCLK	SMBus clock	JTAG2	TCK
6	SMDAT	SMBus data	JTAG3	TDI
7	GND	Ground	JTAG4	TDO
8	+3.3v	+3.3 volt power	JTAG5	TMS
9	JTAG1	+TRST#	+3.3v	+3.3 volt power
10	3.3Vaux	3.3v volt power	+3.3v	+3.3 volt power
11	WAKE#	Link Reactivation	PWRGD	Power Good

🟦 🟥 🟩

CUDA PE Power Model


The real computation problems, Numerical methodologies.
Algorithm Optimization
High Level Code (C CUDA etc)
Coding Optimization
Intermediate Code
Object Code
Executable
Memory operations
CUDA Kernel Object Code
Executable
Operating System
CPU Power Consumption
Memory Power Consumption
GPU Power Consumptior
$E_{CPU} = P_{CPU} t_{CPU}$
$E_{Memory} = P_{memory} t_{memory}$
$E_{GPU} = P_{GPU} t_{GPU}$
$E_{total} = E_{CPU} + E_{GPU} + E_{data} = T(P_{CPU} + P_{GPU} + P_{data})$



Abstract:

Capturing the power characters of each component, building up power model, estimating and validating the power consumption of CUDA PE in SIMD computations.

Method:

1. **CPU power Measurement.** From CPU socket on main board, one approximate way is to measure the CPU input current and voltage at the 8-pin power plug. (Most of the onboard CPUs are powered only by this type of connector)
2. **GPU power measurement.** (Suda paper)
3. **Memory and main board power estimation.** we can make an approximation on its power by measuring the power change on the main board.

$$P_{total}(w) = \sum_{i=1}^{N} P_{GPU}^{i}(w^{i}) + \sum_{j}^{M} P_{CPU}(w^{j}) + P_{mainboard}(w)$$

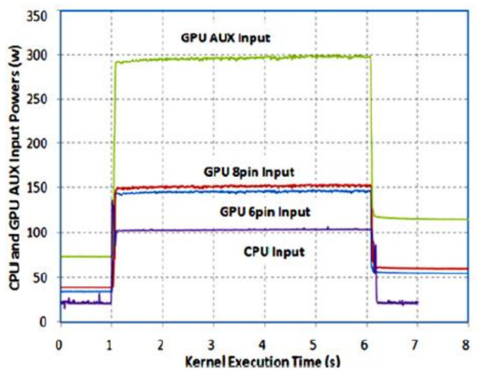
Results:

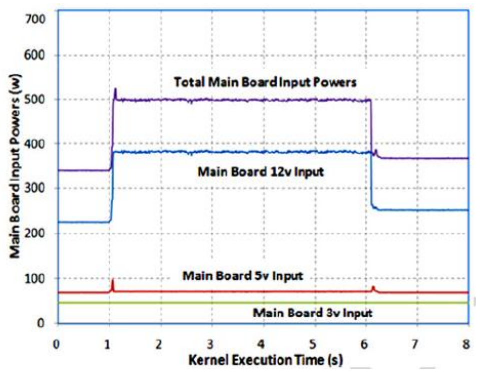
When the matrix size is greater than 1000, the power measurements and program time costs are fairly agree with each other.

Environment:

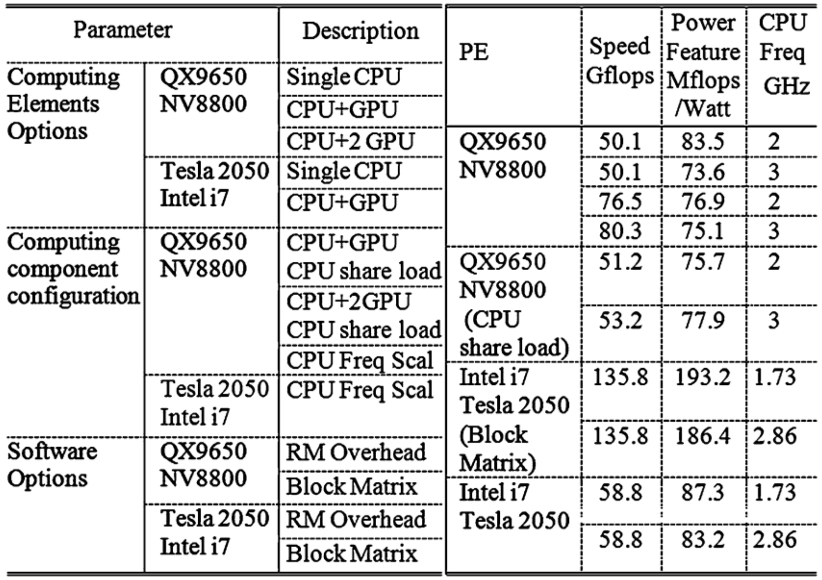
CPU: QX9650 (4cores)/Intel i7 (8cores); Fedora 8/ Ubundu 8; 8GB/3GB DDR3 memory; NVIDIA8800 GTS/640M; 8800GTS512.

CPU-GPU PE Power Feature Determination





Sample on Tesla 1060



Parameter		Description
Computing Elements Options	QX9650 NV8800	Single CPU
		CPU+GPU
		CPU+2 GPU
	Tesla 2050 Intel i7	Single CPU
		CPU+GPU
Computing component configuration	QX9650 NV8800	CPU+GPU CPU share load
		CPU+2GPU CPU share load
		CPU Freq Scal
	Tesla 2050 Intel i7	CPU Freq Scal
Software Options	QX9650 NV8800	RM Overhead
		Block Matrix
	Tesla 2050 Intel i7	RM Overhead
		Block Matrix

PE	Speed Gflops	Power Feature Mflops /Watt	CPU Freq GHz
QX9650 NV8800	50.1	83.5	2
	50.1	73.6	3
	76.5	76.9	2
	80.3	75.1	3
QX9650 NV8800 (CPU share load)	51.2	75.7	2
	53.2	77.9	3
Intel i7 Tesla 2050 (Block Matrix)	135.8	193.2	1.73
	135.8	186.4	2.86
Intel i7 Tesla 2050	58.8	87.3	1.73
	58.8	83.2	2.86

Power features of different PE configurations

Abstract:
Experimental method for estimating component power to build up CUDA PE power model in SIMD computation.

Method:

1.Measuring the power from each component of the PE;

2.Find FLOPS/Watt ratio of the PE to this computation;

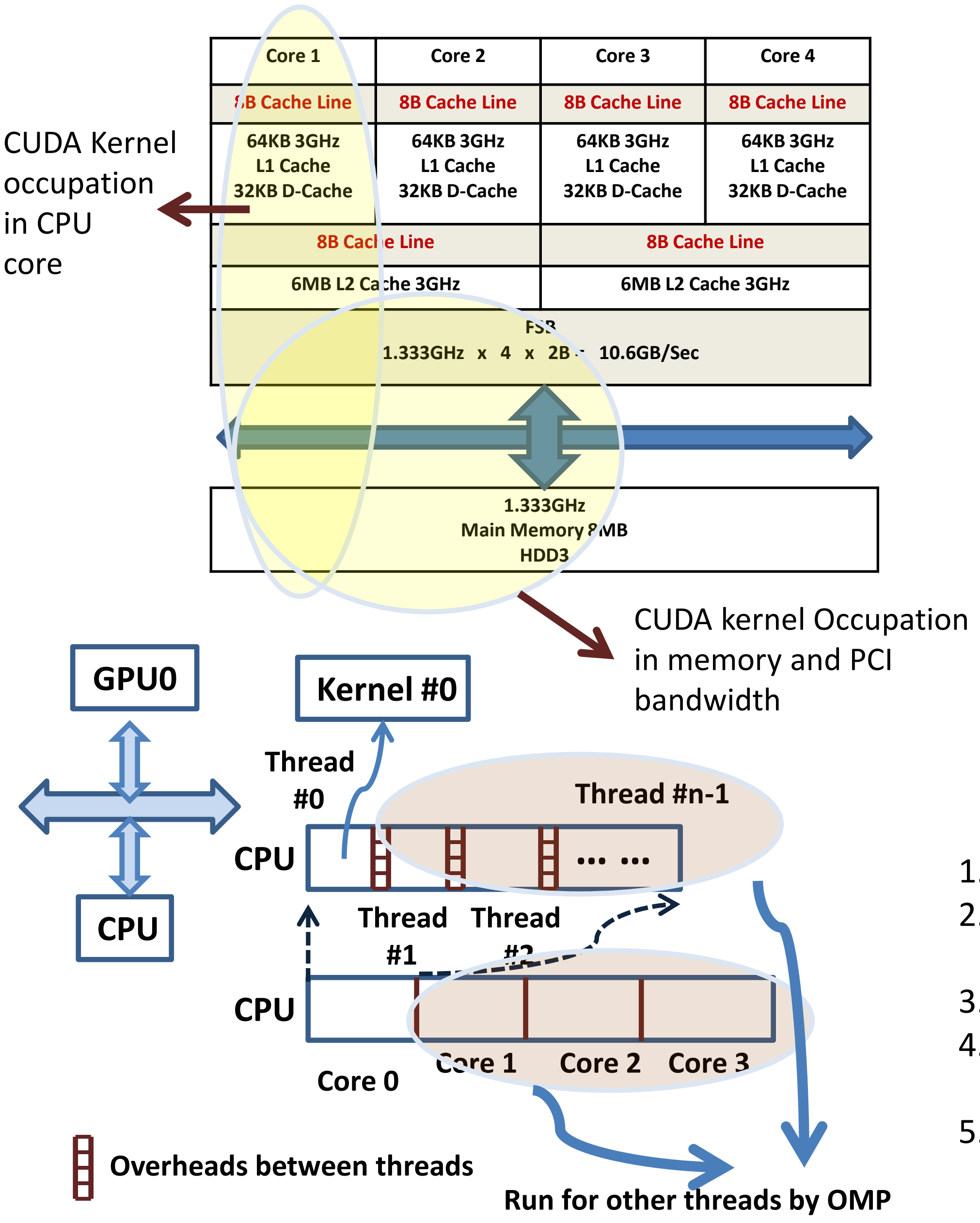
3.Estimated execution time is the total workload FLOP to be computed divides by the computational speed that the CPU-GPU processing element can support;

4.Estimated energy consumption for completing the program is the summation of products of the component powers and the execution times.

Results:
The accuracy of the power model is within 5% percentage error when problem size greater than a threshold of 4000.

Environment:
CUDA PE includes Intel QX9650 CPU/8GB DDR3 memory; GeForce 8800 GTS GPU; OS Fedora 8.

CUDA/OMP Single CUDA device programming model


Core 1
Core 2
Core 3
Core 4
8B Cache Line
64KB 3GHz L1 Cache 32KB D-Cache
8B Cache Line
6MB L2 Cache 3GHz
FSB 1.333GHz x 4 x 2B = 10.6GB/Sec
1.333GHz Main Memory 8MB HDD3
CUDA Kernel occupation in CPU core
CUDA kernel Occupation in memory and PCI bandwidth
GPU0
CPU
Kernel #0
Thread #0
Thread #n-1
CPU
Thread #1
Thread #2
CPU
Core 0
Core 1
Core 2
Core 3
Overheads between threads
Run for other threads by OMP

```
#include <omp.h>
Init CUDA
…
   Kernel ()
   cudaGetDeviceProperties
   cudaSetDevice(i);
   cudaMemset
   cudaMemcpy2D
…
   OMP Thread :
   struct thread_data {
     int thread_id;
     int gpu_id;
     int num_gpus;
     };
…
    struct thread_data *my_data;
    my_data = (struct thread_data *) threadid;
    cpu_thread_id =  my_data->thread_id;
    gpuid = my_data->gpu_id;
    num_gpus = my_data ->num_gpus;
```

1. Setup thread/multi threads;
2. Reserve an individual memory space for CUDA;
3. Bond one thread to CUDA Kernel;
4. Run CUDA kernel by transfer the defined structure;
5. Run other thread as normal OMP threads.

Power performance Improvement by numerical method optimization

$$\mathbf{C} = \mathbf{A}\mathbf{B} \qquad \mathbf{A}, \mathbf{B}, \mathbf{C} \in R^{2^n \times 2^n}$$

If the matrices A, B are not of type $2^n \times 2^n$ we fill the missing rows and columns with zeros.

We partition A, B and C into equally sized block matrices

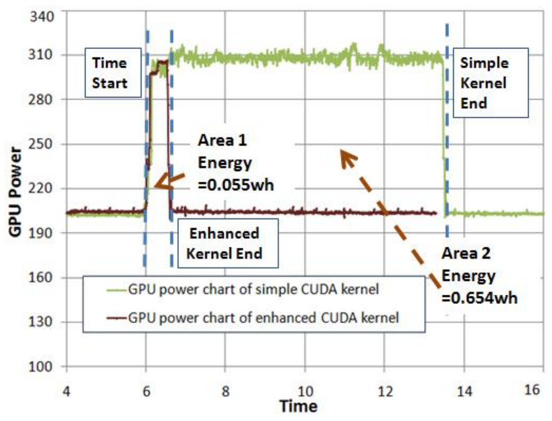
$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

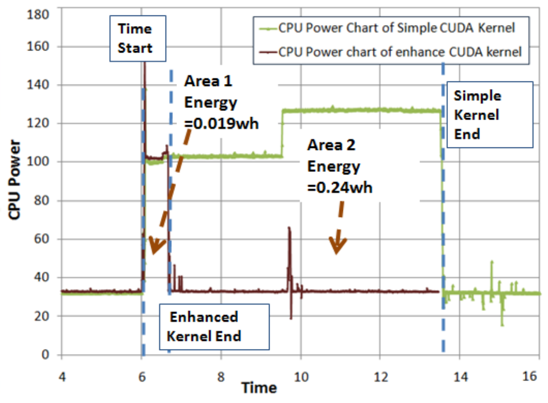
with

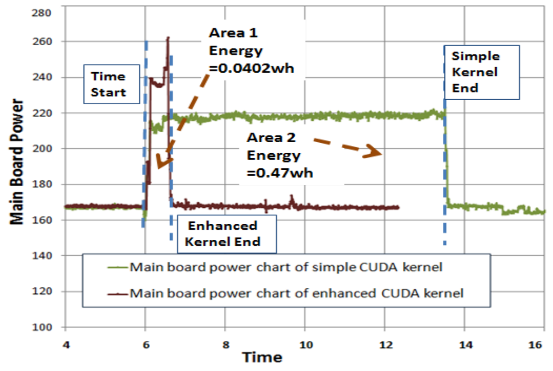
$$\mathbf{A}_{i,j}, \mathbf{B}_{i,j}, \mathbf{C}_{i,j} \in R^{2^{n-1} \times 2^{n-1}}$$

then

$$\begin{aligned} \mathbf{C}_{1,1} &= \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} \\ \mathbf{C}_{1,2} &= \mathbf{A}_{1,1}\mathbf{B}_{1,2} + \mathbf{A}_{1,2}\mathbf{B}_{2,2} \\ \mathbf{C}_{2,1} &= \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1} \\ \mathbf{C}_{2,2} &= \mathbf{A}_{2,1}\mathbf{B}_{1,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2} \end{aligned}$$







	Simple Algorithm	Enhance Algorithm
CPU Energy Consumption	0.24 wh	0.019 wh
GPU Energy Consumption	0.654 wh	0.055 wh
Main Board (Main Memory) Energy Consumption	0.47 wh	0.0402 wh
Overall Time Consumption	7.5 s	0.6s
Overall Energy Consumption	1.364 wh	0.1142 wh

Abstract:

1) Abstract a power model incorporates physical power constrains of hardware;
2) Using block matrices to enhance PCI bus utilization to improve computation performance and save computation power.

Method:

$$P_{total}(w) = \sum_{i=1}^{N} P_{GPU}^{i}(w^{i}) + \sum_{j}^{M} P_{CPU}(w^{j}) + P_{mainboard}(w)$$

Partition smaller matrix-blocks whose size k fits the shared memory in one GPU block. Each GPU block can individually multiply matrix-blocks using its shared memory.

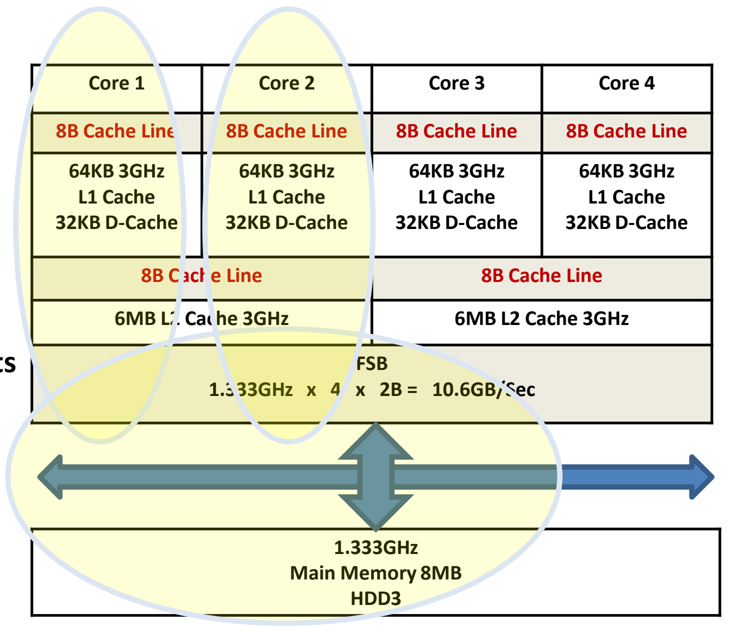
Reduce the data transmission between GPU and main memory to 1/k, will significantly enhance the GPU performance and power efficiency.

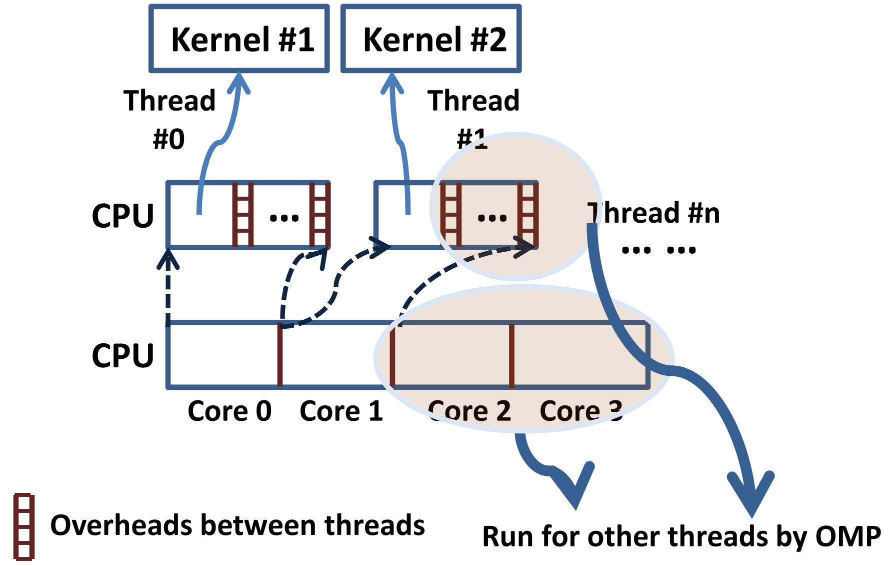
Results: Speedup the overall execution time of simple kernel by 10.81 times, save 91% of energy used by the original kernel.

Environment: Intel core i7 (4cores/8threads); bundu8; 3G DDR3 memory; GPU 8800GTS/640M.

CUDA / OMP multiple GPU device programming model I

Power consuming components

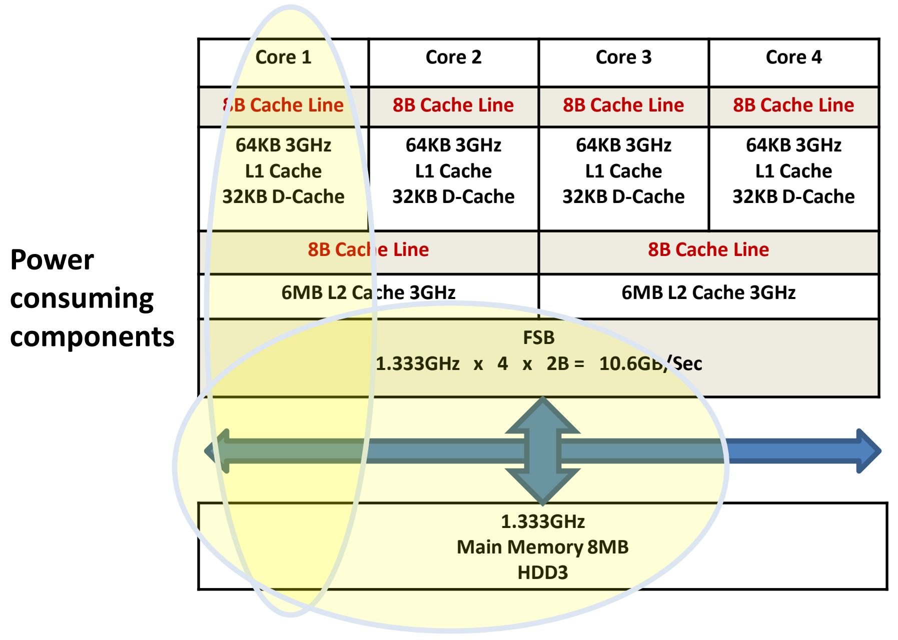




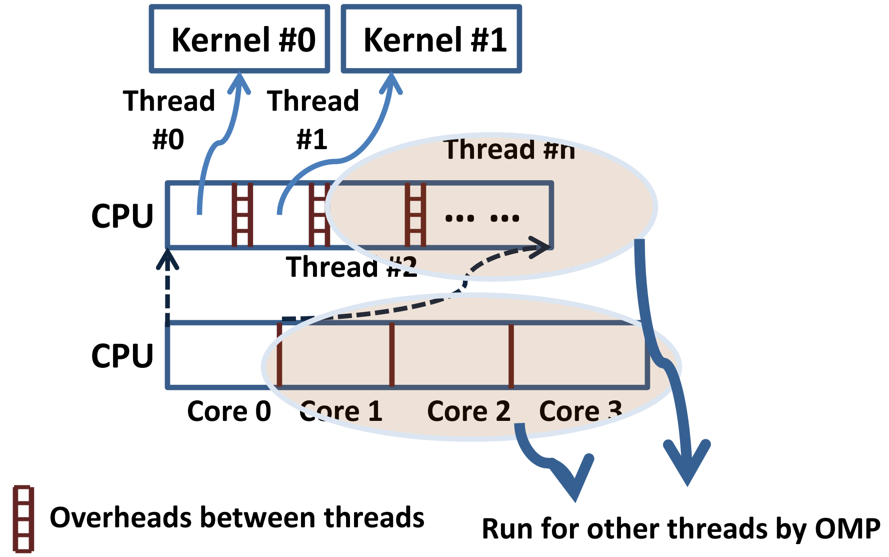
```
#include <omp.h>
Init CUDA
…
    Kernel ()
    cudaGetDeviceProperties
    cudaSetDevice(i);
    cudaMemset
    cudaMemcpy2D
…
CUDA_SAFE_CALL(cudaSetDevice(cpu_thread_id %
num_gpus)); CUDA_SAFE_CALL(cudaGetDevice(&gpu_id))
…
    OMP Thread :
    struct thread_data {
      int thread_id;
      int gpu_id;
      int num_gpus;
      };
…
     struct thread_data *my_data;
     my_data = (struct thread_data *) threadid;
     cpu_thread_id = my_data->thread_id;
     gpuid = my_data->gpu_id;
     num_gpus = my_data ->num_gpus;
```

1. Setup thread/multiple threads;
2. Reserve an individual memory space for CUDA;
3. Bond two threads between two cores and two CUDA devices, respectively;
1. Run CUDA kernels by transferring the defined structure;
2. Run other thread as normal OMP threads.

CUDA / OMP multiple CUDA device programming model II

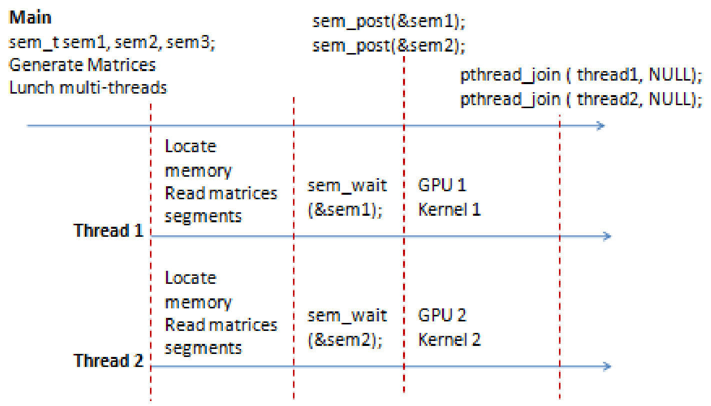

Power consuming components
Core 1
Core 2
Core 3
Core 4
8B Cache Line
8B Cache Line
8B Cache Line
8B Cache Line
64KB 3GHz L1 Cache 32KB D-Cache
64KB 3GHz L1 Cache 32KB D-Cache
64KB 3GHz L1 Cache 32KB D-Cache
64KB 3GHz L1 Cache 32KB D-Cache
8B Cache Line
8B Cache Line
6MB L2 Cache 3GHz
6MB L2 Cache 3GHz
FSB
1.333GHz x 4 x 2B = 10.6Gb/Sec
1.333GHz
Main Memory 8MB
HDD3

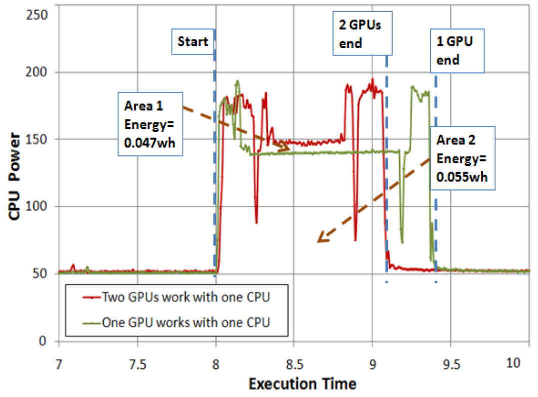
```
#include <omp.h>
Init CUDA
…
    Kernel ()
    cudaGetDeviceProperties
    cudaSetDevice(i);
    cudaMemset
    cudaMemcpy2D
…
CUDA_SAFE_CALL(cudaSetDevice(cpu_thread_id % num_gpus)); CUDA_SAFE_CALL(cudaGetDevice(&gpu_id))
…
    OMP Thread :
    struct thread_data {
      int thread_id;
      int gpu_id;
      int num_gpus;
      };
…
     struct thread_data *my_data;
     my_data = (struct thread_data *) threadid;
     cpu_thread_id =  my_data->thread_id;
     gpuid = my_data->gpu_id;
     num_gpus = my_data ->num_gpus;
```

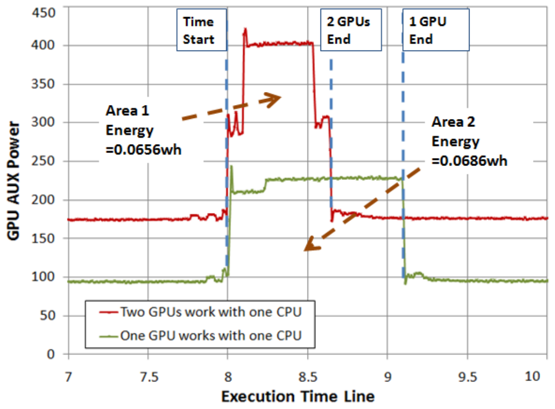


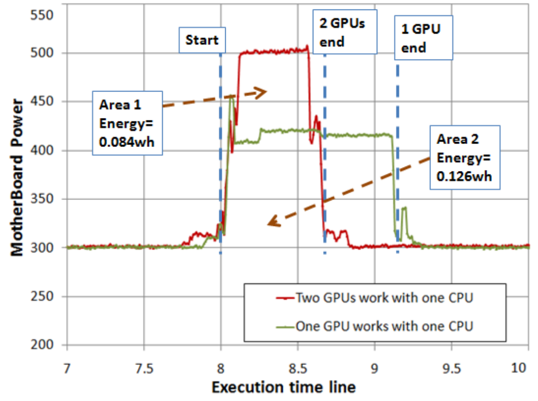
1. Setup thread/multiple threads;
2. Reserve an individual memory space for CUDA;
3. Bond two threads to two CUDA devices, respectively;
4. Run CUDA kernels by transferring the defined structure;
5. Run other thread as normal OMP threads.

Parallel GPU and process synchronization









	CPU+GPU	CPU+2GPUs
CPU Energy Consumption	0.055 wh	0.047 wh
GPU Energy Consumption	0.0686 wh	0.0656 wh
Main Board (Main Memory) Energy Consumption	0.126 wh	0.084 wh
CPU computation time	1.4 s	1.1 s
GPU Kernel computation time	1.1 s	0.65 s
Overall Time Consumption	1. 4 s	1.1 s
Overall Energy Consumption	0.2496 wh	0.1966 wh

Abstract:
Parallel GPU approach with signal synchronization mechanism design; Multithreading GPU kernel control method to save CPU core numbers.

Method:
Partition matrix A into sub-matrices for each GPU device;

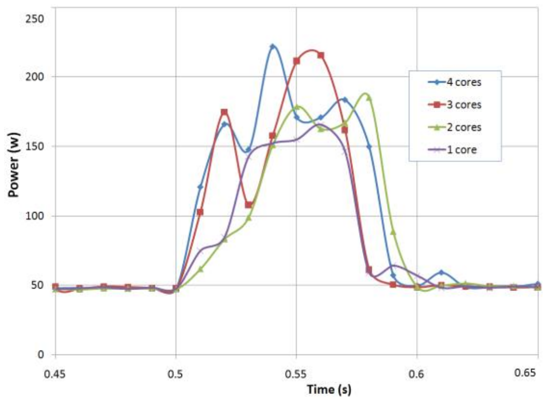
Create multithreads on CPU side to instruct each CUDA kernel;

Design synchronization signal to synchronize each CUDA kernel.

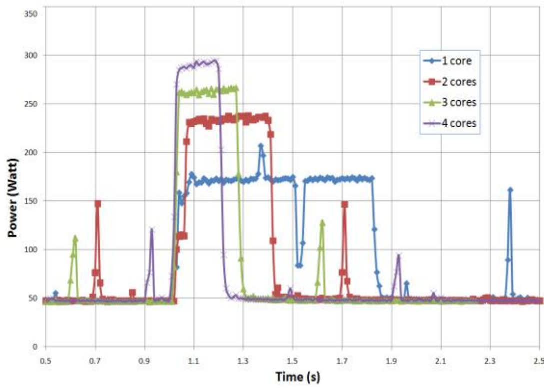
Results:
Parallel GPUs can achieve 71% speedup in Kernel time, 21.4% in CPU time; Power consumption decreased 22%.

Environment:
CUDA PE includes Intel QX9650 CPU/8GB DDR3 memory; GeForce 8800 GTS 512; OS Fedora 8.

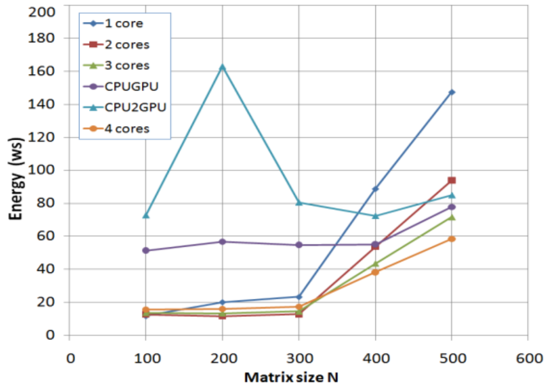
Removing CUDA Overhead



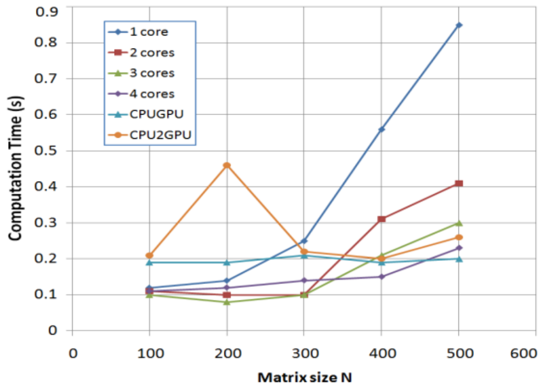
(a)



(b)



(c)



(d)

CUDA computation overhead when workload is mall:
(a) matrix size n=100;
(b) matrix size=500;
(c) Energy cost comparison of 1 to 4 cores, one-GPU PE and two GPU PE;
(d) Computing time comparison of 1 to 4 cores, one-GPU PE and two-GPU PE.

Abstract:

Remove CUDA overhead by calling C function to compute small size workload, save the time and energy cost by CUDA overhead .

Method:

A CUDA overhead for kernel initialization, memory copy and kernel launch before start real kernel computation. A threshold can be determined by experiment by analysis as following:

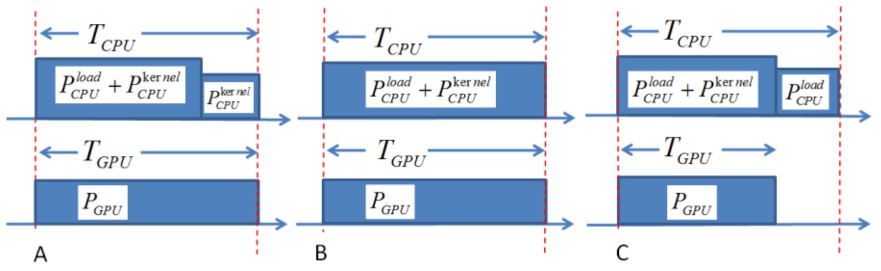
$$T_{CPU}^{k} \le T_{GPU}^{k} = T_{GPUoverhead}^{k} + T_{CUDAkernel}^{k}$$
$$E_{CPU}^{k} = P_{CPU} \times T_{CPU}^{k}$$
$$E_{GPU}^{k} = P_{CPU-GPU-PE} \times T_{GPU}^{k}$$

C function will be slected when matrix size less than k where $E_{CPU}^{k} \le E_{GPU}^{k}$.

Environment:

CUDA PE includes Intel QX9650 CPU/8GB DDR3 memory; GeForce 8800 GTS GPU; OS Fedora 8.

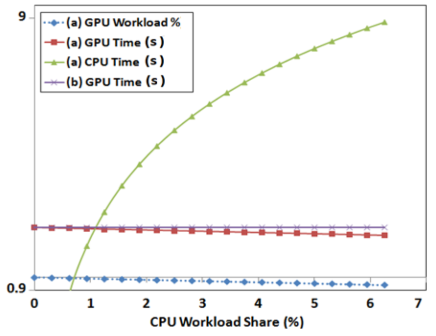
CPU sharing GPU workload

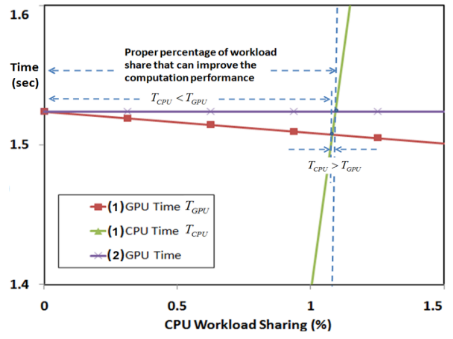


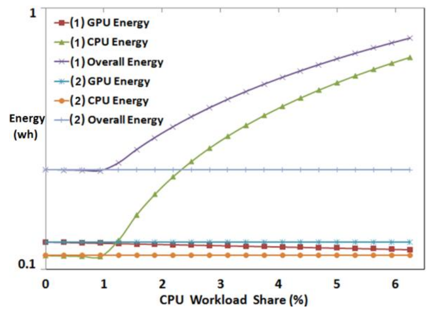
$$A)\ E_{total} = T_{total}(P_{GPU} + P_{CPU}^{kernel}) + T_{CPU}^{load} P_{CPU}^{load} \quad \text{when } T_{GPU} > T_{CPU}$$

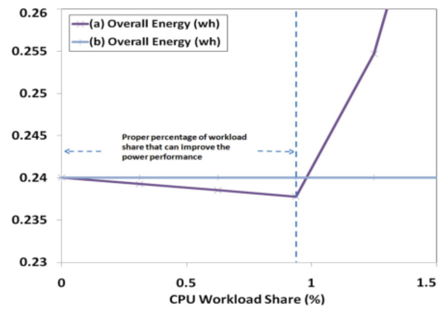
$$B)\ E_{total} = T_{total}(P_{GPU} + P_{CPU}^{load} + P_{CPU}^{kernel}) \quad \text{when } T_{GPU} = T_{CPU}$$

$$C)\ E_{total} = T_{GPU}(P_{GPU} + P_{CPU}^{load} + P_{CPU}^{kernel}) + (T_{total} - T_{GPU})P_{CPU}^{load} \quad \text{when } T_{GPU} < T_{CPU}$$









Abstract:

Determine the load to be shared by CPU based on the computation character and performance estimation.

Method:

$$T_{CPU} = \frac{W_{CPU}}{s_{CPU}},\ T_{GPU} = \frac{W_{GPU}}{s_{GPU}}$$

$$T = \max(T_{CPU}, T_{GPU})$$

$$E_{CPU} = T \cdot P_{CPU};\ E_{GPU} = T_{GPU} \cdot P_{GPU} = \frac{W_{GPU}}{f_{GPU}}$$

$$E = E_{CPU} + E_{GPU}$$

$$E_{\min} = (E_{CPU} + E_{GPU})_{\min} = (T \cdot P_{CPU} + T_{GPU} \cdot P_{GPU})_{\min}$$

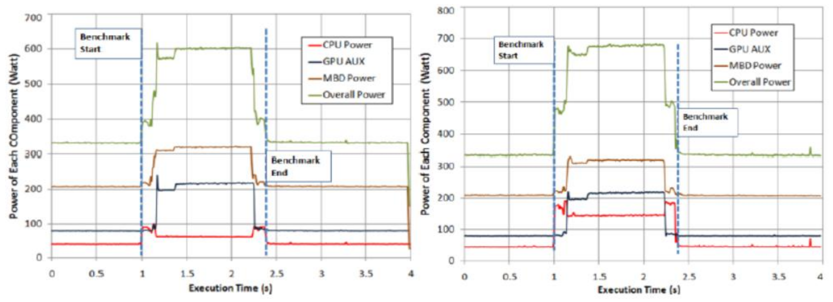
Results:

An optimized minimum energy value can be obtained when CPU (one core) workload share is around 0.83%, the maximum energy saving can reach around 1.3%. (for devices listed below)

Environment:

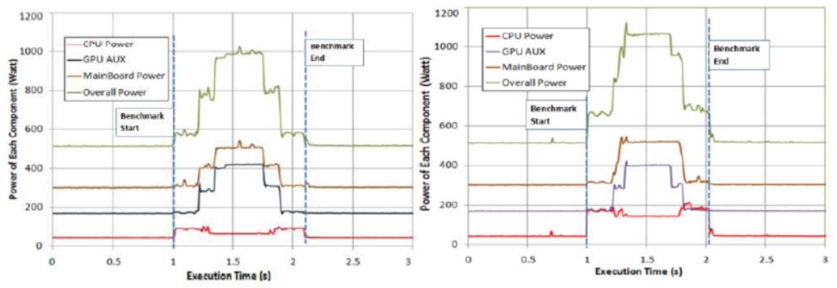
CUDA PE includes Intel QX9650 CPU/8GB DDR3 memory; GeForce 8800 GTS GPU; OS Fedora 8.

CPU Frequency Scaling



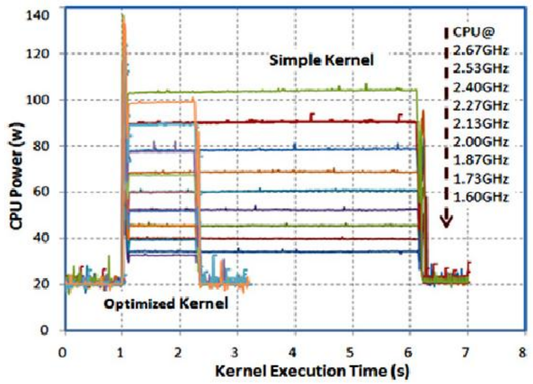
Power chart of CUDA on QX9650 (running on 2GHz) and GF 8800 GST/512 GPU.

Power chart of CUDA on QX9650 (running on 3GHz) and GF 8800 GST/512 GPU.



Power chart of CUDA on QX9650 (running on 2GHz) and 2 GF 8800 GST/512 GPUs.

Power chart of CUDA on QX9650 (running on 3GHz) and 2 GF 8800 GST/512 GPUs



Abstract:
Design a CPU frequency scaling method to save CUDA PE power without decreasing the computation performance.

Method:
CPU frequency should match CUDA kernel calls in order to not decrease GPU computation speed.

CPU frequency can be scaled down without compromising with the PE's performance however to save the CPU's power.

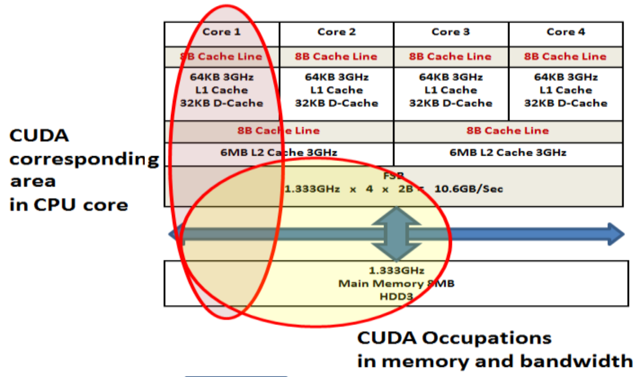
A rough estimation for the minimum CPU frequency should be satisfy

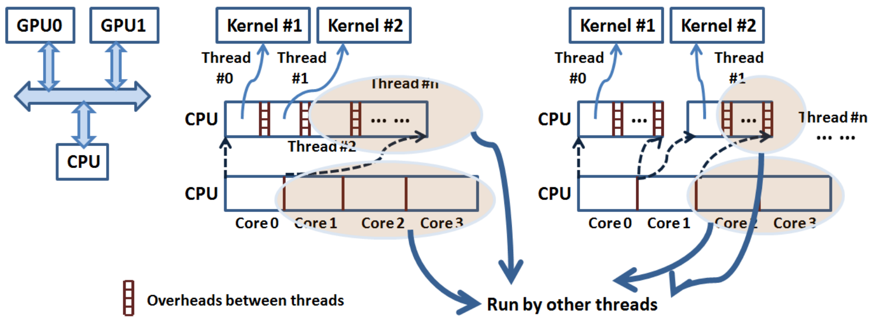
$$F_{CPU} \geq F_{CPI} \text{ (most of the cases)}$$
$$F_{CPU} \geq F_{GPUMemory} \text{ (if } F_{CPI} \geq F_{GPUMemory}\text{)}$$

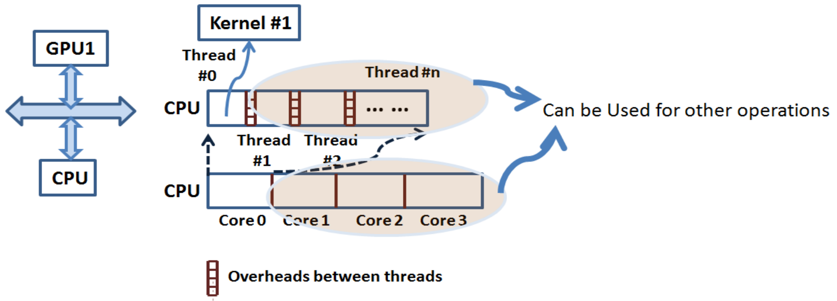
Results:
An optimized minimum energy value can be obtained when CPU runs in low frequency (2GHz), comparing with CPU in 3GHz the total PE energy saving can reach 12.43% in average when matrix size increases from 500 to 5000, without computation speed decrease.

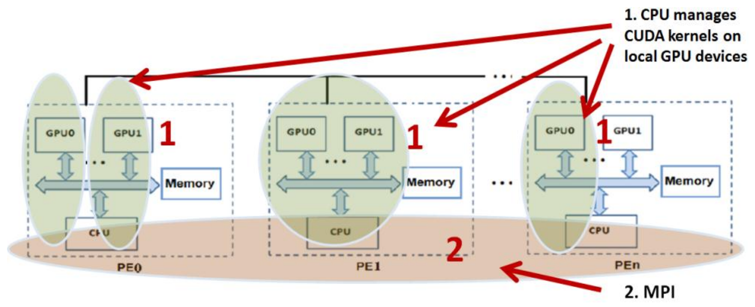
Environment:
CUDA PE includes Intel QX9650 CPU/8GB DDR3 memory; GeForce 8800 GTS GPU; OS Fedora 8.

CUDA / MPI load scheduling for energy aware computing


Core 1
Core 2
Core 3
Core 4
8B Cache Line
64KB 3GHz L1 Cache 32KB D-Cache
8B Cache Line
6MB L2 Cache 3GHz
FSB
1.333GHz x 4 x 2B = 10.6GB/Sec
1.333GHz Main Memory 8MB HDD3
CUDA corresponding area in CPU core
CUDA Occupations in memory and bandwidth



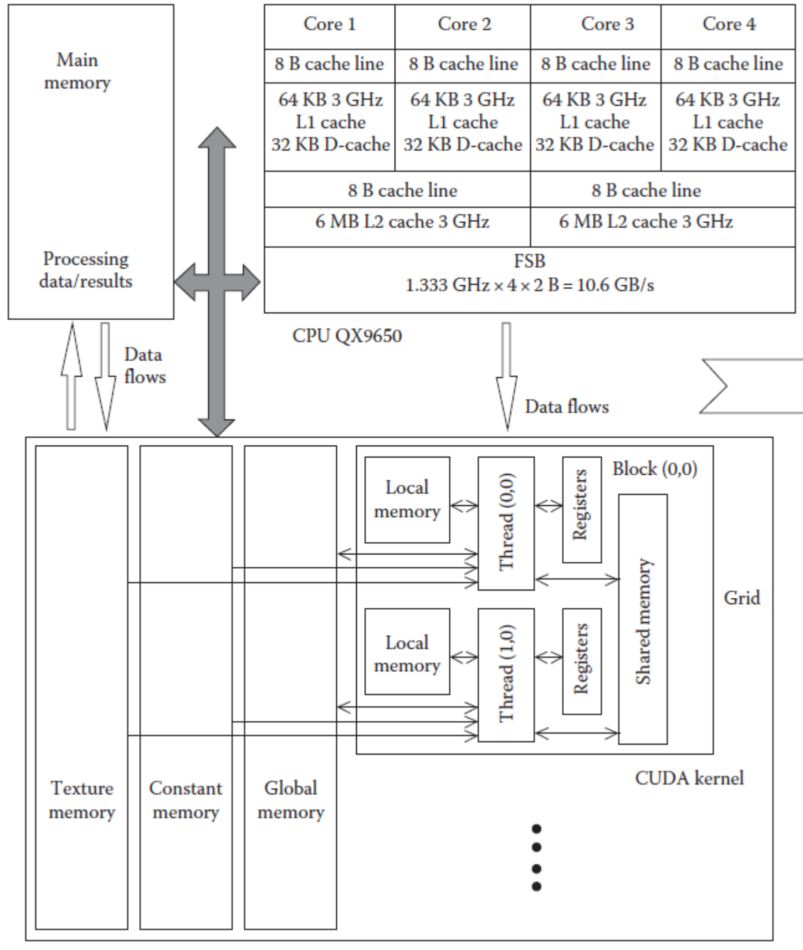




Abstract:
With C/CUDA/MPI on Multi-core and GPU clusters, partitioning and scheduling SPMD and SIMD program to Multi-core CPU and GPU cooperative architectures .
MPI works as data distributing mechanism between the GPU nodes and CUDA as the computing engine.
Method:
Multi complier , MPI cluster computing algorithms and communication strategies are involved.
Environment:
CUDA PE includes Intel QX9650 CPU/8GB DDR3 memory; GeForce 8800 GTS GPU; OS Fedora 8.

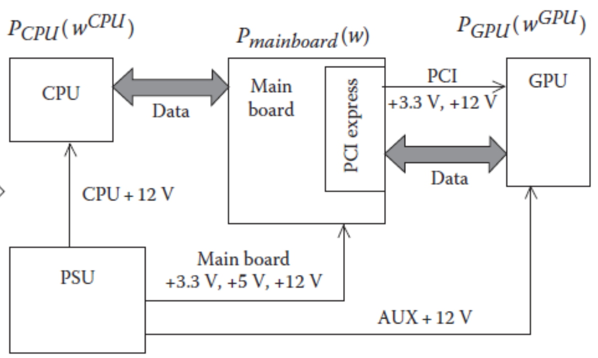
H/S power performance factors for global Optimization

GPU power model



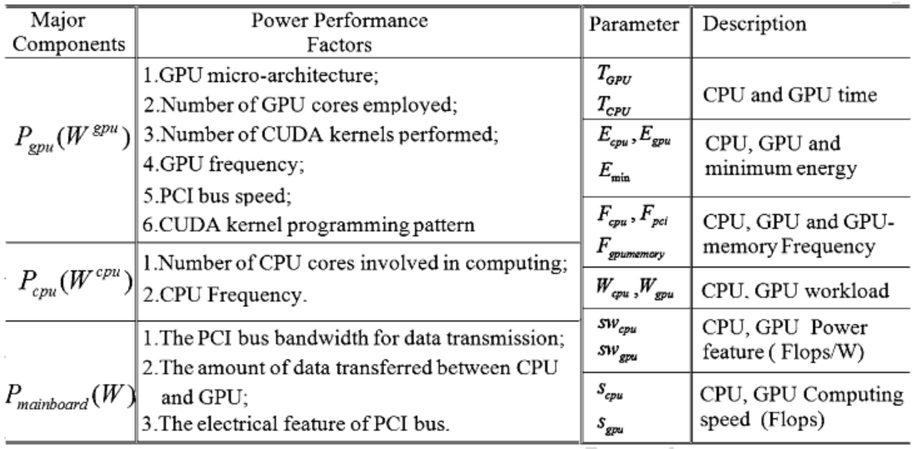
(a) GPU NVIDIA 8800 GTS

GPU and multicore power-consuming component abstractions



$$P_{system}(w) = P_{GPU}(w^{GPU}) + P_{CPU}(w^{CPU}) + P_{mainboard}(w)$$

(b)



Major Components	Power Performance Factors

$P_{gpu}(W^{gpu})$	1.GPU micro-architecture;
	2.Number of GPU cores employed;
	3.Number of CUDA kernels performed;
	4.GPU frequency;
	5.PCI bus speed;
	6.CUDA kernel programming pattern

$P_{cpu}(W^{cpu})$	1.Number of CPU cores involved in computing; 2.CPU Frequency.
$P_{mainboard}(W)$	1.The PCI bus bandwidth for data transmission; 2.The amount of data transferred between CPU and GPU; 3.The electrical feature of PCI bus.

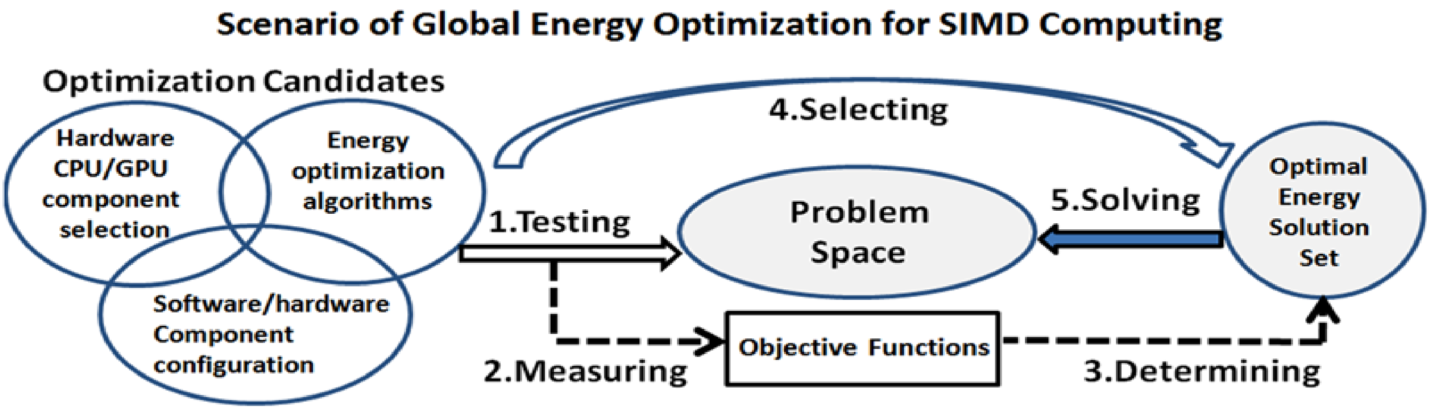
Parameter	Description
T_{GPU} T_{CPU}	CPU and GPU time
E_{cpu}, E_{gpu} E_{min}	CPU, GPU and minimum energy
F_{cpu}, F_{pci} $F_{gpumemory}$	CPU, GPU and GPU-memory Frequency
W_{cpu}, W_{gpu}	CPU. GPU workload
sw_{cpu} sw_{gpu}	CPU, GPU Power feature (Flops/W)
s_{cpu} s_{gpu}	CPU, GPU Computing speed (Flops)

Energy parameters

Scenario of global Energy Optimization for SIMD Computing

Definition		Description
Problem Space		The multiplication for variable length of dense matrices and , with multicore and GPU(s) device.
Optimization Candidates	**Hardware Components**	Selection and employment of the number of CPUs and GPUs for solving the problem.
	Component Configurations	Frequency scaling on CPU and/or GPU components.
	Optimization Algorithms	The optimization algorithm designed and implemented for solving the problem that available for optimizer to choose. Including parallelization scheme and workload scheduling.
Objective Functions		The objective function which measure the utility of the solution candidates to find the minimization.
Optimal Solution set		Determine the number of components to be included in the final solution so that the total time is less than or equal to a given limit and the total energy is as minimum as possible.

Definitions of global optimization model

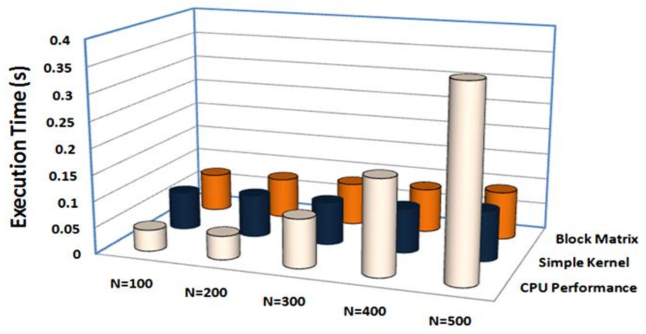


Scenario of Global Energy Optimization for SIMD Computing
Optimization Candidates
Hardware CPU/GPU component selection
Energy optimization algorithms
Software/hardware Component configuration
4.Selecting
1.Testing
Problem Space
5.Solving
Optimal Energy Solution Set
Objective Functions
2.Measuring
3.Determining

Global optimizations

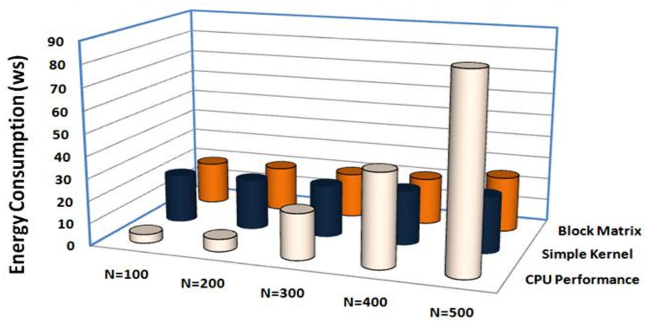
Numerical approach + Parallel GPU + Load scheduling

Computation Performance of Small Size Matrices Multiplication



	N=100	N=200	N=300	N=400	N=500
□ CPU Performance	0.041	0.045	0.093	0.181	0.361
■ Simple Kernel	0.073	0.081	0.082	0.085	0.095
🟧 Block Matrix	0.073	0.077	0.081	0.084	0.092

Energy Consumptions of Small Size Matrices Multiplication

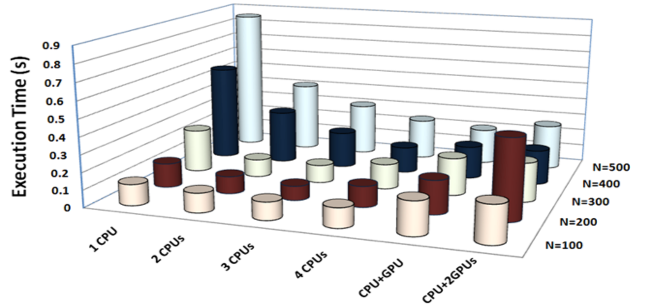


	N=100	N=200	N=300	N=400	N=500
□ CPU Performance	3.98	5.58	20.71	41.76	86.12
■ Simple Kernel	21.78	22.64	23.33	24.44	25.34
🟧 Block Matrix	19.08	19.89	20.11	21.02	24.78

The energy consumption on computing the multiplications of small matrices of size 100 to 500 using one multicore with 4 cores / 8 threads (Intel i7) and one GPU (Tesla 2050C), with simple Kernel and block matrix, respectively.

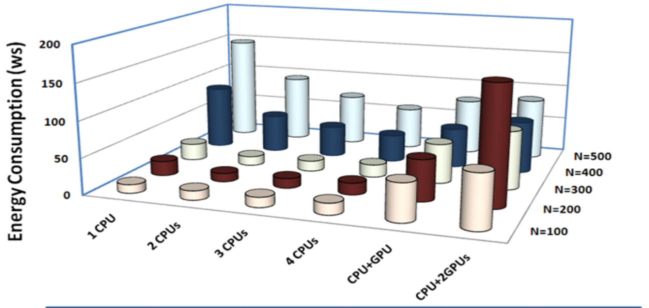
Remove CUDA overhead + Parallel GPU + Load scheduling

Computation Performance on Different Components



	1 CPU	2 CPUs	3 CPUs	4 CPUs	CPU+GPU	CPU+2GPUs
□ N=100	0.122	0.112	0.103	0.113	0.192	0.215
■ N=200	0.141	0.101	0.083	0.121	0.194	0.463
□ N=300	0.252	0.104	0.105	0.146	0.217	0.227
■ N=400	0.564	0.312	0.212	0.153	0.191	0.202
□ N=500	0.853	0.411	0.305	0.238	0.208	0.264

Energy Consumptions on Different Components



	1 CPU	2 CPUs	3 CPUs	4 CPUs	CPU+GPU	CPU+2GPUs
□ N=100	11.82	12.73	13.69	15.73	51.46	72.86
■ N=200	20.24	11.73	13.41	16.24	56.85	163.05
□ N=300	23.39	12.84	14.57	17.43	54.87	80.55
■ N=400	88.87	53.01	43.49	38.26	55.21	72.48
□ N=500	147.48	93.88	71.53	58.48	77.83	85.01

The energy consumption on the same problems using one to four cores (QX9650), one-CPU-one-GPU(8800GTS) CUDA PE and one-CPU-two-GPU(8800GTS) CUDA PE, respectively.

Conclusion and future work

Conclusion

1. An experimental power modeling and estimation method on GPU and multicore structures has been illustrated;
2. Power parameters are captured by measurements on each component in a CUDA PE, thus power features to the SIMD program can then be analyzed and obtained;
3. Five energy aware algorithm design methods have been introduced;
4. A global energy optimization model is created for CUDA PE by a four-tuple definition that specifies the problem space, the objective functions, optimization candidates and optimal solution set, the procedure to find optimal energy solution is described based on it.
5. The global energy optimization model is validated by examining C/CUDA programs executing on real systems.

Future work

1. Energy estimation method can be refined to enhance its precision by including more components;
2. Power parameters can be tuned for obtaining the minimum energy consumption for given problems;
3. Global optimization methods can be used on managing energy aware software design constrains in order to reach the best energy performance among all possible alternatives.