CScADS Autotuning Panel
Industrial Software Challenges

Chi-Keung (CK) Luk

Software Pathfinding and Innovation
Intel

Aug 12, 2009
My Background

• Part of the path-finding team at Intel Software:
  – R&D tools for program analysis, architectural studies, web programming, and parallel programming

• My previous projects:
  – The Ispike profile-guided optimizer for Itanium
  – The Pin dynamic binary instrumentation tool

• Current projects:
  – A software system that automatically distributes works on heterogeneous multicores
  – A tool for discovering potential parallelism in serial code
  – An autotuning tool
Profile-Guided Optimization and Autotuning

• Profile-guided optimization (PGO) has been shown effective in reducing branch mispredicts, cache and TLB misses
  – PGO improves Oracle performance by 30% over best non-PGO
    (Kernel Optimizations and Prefetch with the Spike Executable Optimizer, Flower et.al., FDDO 2001).

• However many ISVs do not use PGO in their shipped products (they may use it for benchmarking though).

• ISV’s concerns:
  – Input sensitivity:
    • No guarantee that that training inputs used in driving the PGO are representative of the real inputs
  – Ease of use:
    • Developers don’t’ like to do multiple passes of compilation
    • Need to reprofile even with minor changes to the application

• Autotuning can be viewed as iterative PGO and would face the same challenges as PGO does to get ISVs’ adoption
Interesting Approach: Online Autotuning

• Online autotuning (my definition):
  – An approach with which the application is able to tune itself *after* it is deployed to the field
    • May or may not involve dynamic code generation

• Online autotuning could address the challenges in:
  – Input sensitivity:
    • Dynamically adapt to input and phase changes
  – Ease of use:
    • The entire process of profiling, tuning, and recompilation is totally transparent to the user

• Issue with online autotuning:
  – Reproducibility of bugs
    • Need deterministic replay of the execution
An Optimistic View of Online Autotuning

• Reasons why I think Online Autotuning could be adopted by the software industry:
  – The increasing popularity of dynamic language/programming environments:
    • Java and .Net
    • Scripting languages (Python, Ruby)
    • DirectX
  – Unused cores during runtime can be used to tune the application
Problem:

– Determine the optimal distribution of works on a heterogeneous machine at runtime

Initial Research Results:

Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping.
CK Luk, Sunpyo Hong, Hyesoon Kim.
Heterogeneous Architectures (1)

• A particularly interesting class of parallel machines is **Heterogeneous Architecture:**
  – Multiple types of Processing Elements (PEs) available on the same machine
Heterogeneous Architectures (2)

- Heterogeneous architectures are increasingly popular:
  - Multicore CPU + GPU
  - IBM’s Cell processor
  - Intel’s Larrabee

focus of this talk
Software Challenge

A CPU + GPU system:

The Mapping Problem:

Map computations to PEs to optimize an objective function, which could be:

- Performance
- Energy
- Performance / Energy
Existing Solutions to the Mapping Problem

- Programmer performs the mapping **manually** and **statically**
- Examples:
  - IBM XL compiler extension that supports OpenMP on the Cell
  - Intel CTG’s ExoCHI/Merge framework for programming the CPU and GPU
- Disadvantages:
  - Labor intensive
  - Not adaptable to changes in runtime environments
Case Study: Matrix Multiplication

• Heterogeneous machine used:
  – CPU: dual-socket QuadCore (max = 8 cores)
  – GPU: Nvidia GTX-8800 GPU

• Three configurations tested:
  1. Small problem size, max CPU cores used
  2. Big problem size, max CPU cores used
  3. Big problem size, fewer CPU cores used

• In each configuration:
  – Perform cooperative matrix multiplication
    (varying the distribution of works over the CPU and GPU)
Cooperative Matrix Multiplication

\[ C = A \times B \]

CPU

\[ C_1 = A_1 \times B \]

GPU

\[ C_2 = A_2 \times B \]
Cooperative Matrix Multiplication Results

<table>
<thead>
<tr>
<th>Configuration 1:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix dimension size = 1000</td>
<td>#CPU cores = 8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration 2:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix dimension size = 6000</td>
<td>#CPU cores = 8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration 3:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix dimension size = 6000</td>
<td>#CPU cores = 2</td>
</tr>
</tbody>
</table>

Lessons Learned:

The optimal PE mapping depends on the application, the input size, and hardware resources available.

⇒ Need an automatic and dynamic technique that takes all these factors into account

Our contribution: ADAPTIVE MAPPING
Outline

• Introduction
  • The Qilin Programming System
  • Adaptive Mapping Algorithm
  • Experimental Evaluation
  • Conclusions
The Qilin Parallel Programming System

What is “Qilin”?

• A mythical Chinese creature whose body consisting of parts from different kinds of animals (i.e., heterogeneous!)
Qilin Programming Model

• Based on Stream Programming:
  – Data is represented as contiguous arrays of records (streams)
  – Computations are structured as a series of kernels on streams

• Advantages of this model:
  – Explicit parallelism and data locality

• Applicability:
  – Media applications
  – Scientific apps (regular and irregular)
    • E.g., SpecFP2006, Berkeley dwarfs
The Qilin System

**Qilin API:**
- Express parallelizable computations explicitly

**Compiler:**
- Dynamically compile Qilin API calls into native codes

**Code Cache:**
- Save code previously compiled

**Dev. Tools:**
- Profiler, debugger, visualization

**Libraries:**
- Contain commonly used functions (BLAS and FFT)

**Scheduler:**
- Schedule threads onto CPU and GPU
Prototype Implementation

• **Compiler:**
  – Implement Qilin API on top of high-level threading APIs:
    • CPU: Intel Threading Building Block (TBB)
    • GPU: Nvidia CUDA
  – Invoke native compilers to generate final executables

• **Libraries:**
  – Implemented as wrapper calls to vendor-provided libraries:
    • CPU: Intel Math Kernel Library (MKL)
    • GPU: Nvidia CUDA BLAS (CUBLAS)

• **Scheduler:**
  – Simply rely on the TBB task scheduler
  – Devote 1 CPU thread to handshake with the GPU
Qilin API

- Qilin defines two new classes of objects:
  - **QArray**: a multidimensional array of a generic type
  - **QArrayList**: A list of QArray objects
- All parallel computations are on QArrays or QArrayLists
- Two approaches to expressing parallel computations in Qilin:
  - Using Stream API
  - Using Threading API
The Stream API Approach

• Qilin implements common data-parallel operations:
  – Element-wise
  – Reduction
  – Linear algebra

• Similar to the stream APIs found in GPGPU systems:
  – Brook, Peakstream, Rapidmind, Accelerator, Ct

• However, Qilin allows programmer to select the PE for each operation
void MySgemm(float* A, float* B, float* C, int m, int k, int n,
    float alpha, float beta)
{

    // Create Qilin arrays from normal arrays
    QArray<float> qA = QArray<float>::Create2D(m, k, A);
    QArray<float> qB = QArray<float>::Create2D(k, n, B);
    QArray<float> qC = QArray<float>::Create2D(m, n, C);

    // Invoke the Qilin version of BLAS Sgemm() on the
    // processing elements determined by the default mapping scheme
    qC = BlasSgemm(qA, qB, qC, alpha, beta, PE_SELECTOR_DEFAULT);

    // Convert from qC[] back to C[] and
    // this triggers the lazy evaluation
    qC.ToNormalArray(C, m*n*sizeof(float))
}
The Threading API Approach

• Programmer provides implementations of parallel operations using the underlying threading APIs (i.e., TBB and CUDA)

• Qilin is responsible for:
  – Distributing computations over CPU and GPU
  – Presenting a single result by merging partial results from CPU and GPU

• Example: Image Filter
  – See next slide
void CpuFilter(QArray<Pixel> qSrc, QArray<Pixel> qDst) {
    Pixel* src_cpu = qSrc.NormalArray(), dst_cpu = qDst.NormalArray();
    int height_cpu = qSrc.DimSize(0), width_cpu = qSrc.DimSize(1);

    // … Filter implementation in TBB …
}

void GpuFilter(QArray<Pixel> qSrc, QArray<Pixel> qDst) {
    Pixel* src_gpu = qSrc.NormalArray(), dst_gpu = qDst.NormalArray();
    int height_gpu = qSrc.DimSize(0), width_gpu = qSrc.DimSize(1);

    // … Filter implementation in CUDA …
}

void MyFilter(Pixel* src, Pixel* dst, int height, int width) {

    // Create Qilin arrays from normal arrays
    QArray<Pixel> qSrc = QArray<Pixel>::Create2D(height, width, src);
    QArray<Pixel> qDst = QArray<Pixel>::Create2D(height, width, dst);

    // Define myFilter as an operation that glues CpuFilter() and GpuFilter()
    QArrayOp myFilter = MakeQArrayOp("myFilter", CpuFilter, GpuFilter);

    // Build the argument list for myFilter. QILIN_PARTITIONABLE means the
    // associated computation can be partitioned to run on both CPU and GPU.
    QArrayOpArgsList argList;
    argList.Insert(qSrc, QILIN_PARTITIONABLE);
    argList.Insert(qDst, QILIN_PARTITIONABLE);

    // Apply myFilter with argList using the default mapping scheme
    QArray<BOOL> qSuccess = ApplyQArrayOp(myFilter, argList, PE_SELECTOR_DEFAULT);

    // Convert from qSuccess[] to success, and this triggers the lazy evaluation
    BOOL success;
    qSuccess.ToNormalArray(&success, sizeof(BOOL));
Qilin Dynamic Compilation Process

Steps in the compilation process:

1. Build Directed Acyclic Graphs (DAGs) from Qilin API calls

2. Map computations in DAGs to Processing Elements (PEs)

3. Perform optimizations on the DAGs:
   - Operator coalescing (to increase granularity of parallelization)
   - Removal of unnecessary temporary arrays

4. Code generation
Adaptive Mapping

- A technique to automatically find the near-optimal mapping for the given program, problem size and hardware
- Each <program, hardware> configuration involves one training run and many reference runs:
  - Training run:
    - Qilin finds the execution-time projections of the CPU and the GPU for the given configuration
  - Reference run:
    - Qilin computes the near-optimal distribution of work for the current problem size
Training Run

Training Run

Kernel $K$

Data taken: $T_c(N_{1,1}) \cdots T_c(N_{1,m})$

$T'_c(N) = \text{The projected time to execute the kernel of problem size } N \text{ on the CPU}$

$= a_c + b_c * N$

$T'_g(N) = \text{The projected time to execute the kernel of problem size } N \text{ on the GPU}$

$= a_g + b_g * N$
\( \beta = \) Fraction of work mapped to CPU  
\( p = \) Number of CPU cores  
\( N = \) Problem size

\[ T'_{\beta}(N) = \text{The projected time to execute} \]
\[ \beta N \text{ work on the CPU and} \]
\[ (1- \beta)N \text{ work on the GPU} \]

\[ = \text{Max}( p/(p-1)T'_C(\beta N), \ T'_G((1-\beta)N) ) \]

Once \( N \) is fixed to the actual problem size \( N_r \), we find the \( \beta \) that minimizes \( T'_{\beta}(N_r) \).

We consider where the two curves \( p/(p-1)T'_C(\beta N_r) \) and \( T'_G((1-\beta)N_r) \) intersect.

There are 3 possible cases (see next slide).
Three Possible Cases of $\beta$

**Case i: CPU and GPU curves intersect at $\beta \leq 0$**

- **CPU**: $(p/p-1)T'_c(\beta N_r)$
- **GPU**: $T'_G((1-\beta)N_r)$

Minimized when mapping all work to the GPU

**Case ii: The two curves intersect at $\beta \geq 1$**

- **GPU**: $T'_G((1-\beta)N_r)$
- **CPU**: $(p/p-1)T'_c(\beta N_r)$

Minimized when mapping all work to the CPU

**Case iii: The two curves intersect at $0 < \beta < 1$**

- **GPU**: $T'_G((1-\beta)N_r)$
- **CPU**: $(p/p-1)T'_c(\beta N_r)$

Minimized when mapping $\beta_{min}$ of work to the CPU

Minimized when mapping all work to the GPU
Experimental Framework

Heterogeneous PC used:

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Intel Core2 Quad</td>
<td>Nvidia 8800 GTX</td>
</tr>
<tr>
<td>Core Clock</td>
<td>2.4 GHz</td>
<td>575 MHz</td>
</tr>
<tr>
<td>Number of Cores</td>
<td>8 cores (on 2 sockets)</td>
<td>128 stream processors</td>
</tr>
<tr>
<td>Memory Size</td>
<td>4 GB</td>
<td>768 MB</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>8 GB/s</td>
<td>86.4 GB/s</td>
</tr>
<tr>
<td>Threading API</td>
<td>Intel TBB</td>
<td>Nvidia CUDA</td>
</tr>
<tr>
<td>Compiler</td>
<td>ICC 10.1</td>
<td>NVCC 1.1</td>
</tr>
<tr>
<td>OS</td>
<td>32-bit Linux Fedora Core 6</td>
<td></td>
</tr>
</tbody>
</table>
**Benchmarks**

- A set of important computation kernels:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binomial</td>
<td>American option pricing</td>
<td>CUDA SDK</td>
</tr>
<tr>
<td>BlackScholes</td>
<td>European option pricing</td>
<td>CUDA SDK</td>
</tr>
<tr>
<td>Convolve</td>
<td>2D separable image convolution</td>
<td>CUDA SDK</td>
</tr>
<tr>
<td>MatrixMultiply</td>
<td>Dense matrix multiplication</td>
<td>CUDA SDK</td>
</tr>
<tr>
<td>Linear</td>
<td>Linear image filter---compute output pixel as average of a 9-pixel square</td>
<td>Intel’s Merge</td>
</tr>
<tr>
<td>Sepia</td>
<td>Modify RGB value to artificially age images</td>
<td>Merge</td>
</tr>
<tr>
<td>Smithwat</td>
<td>Compute scoring matrix for a pair of DNA sequences</td>
<td>Merge</td>
</tr>
<tr>
<td>Svm</td>
<td>Kernel from a SVM-based face classifier</td>
<td>Merge</td>
</tr>
</tbody>
</table>
Adaptive mapping achieves 94% of the speedup of manual mapping
Adaptive mapping is nearly as good as manual mapping in energy consumption
## Distribution of Computations

<table>
<thead>
<tr>
<th></th>
<th>Manual mapping</th>
<th></th>
<th>Adaptive mapping</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU</td>
<td>GPU</td>
<td>CPU</td>
<td>GPU</td>
</tr>
<tr>
<td>Binomial</td>
<td>10%</td>
<td>90%</td>
<td>10.5%</td>
<td>89.5%</td>
</tr>
<tr>
<td>BlackScholes</td>
<td>40%</td>
<td>60%</td>
<td>46.5%</td>
<td>53.5%</td>
</tr>
<tr>
<td>Convolve</td>
<td>40%</td>
<td>60%</td>
<td>36.3%</td>
<td>63.7%</td>
</tr>
<tr>
<td>MatrixMultiply</td>
<td>40%</td>
<td>60%</td>
<td>45.5%</td>
<td>54.5%</td>
</tr>
<tr>
<td>Linear</td>
<td>60%</td>
<td>40%</td>
<td>50.8%</td>
<td>49.2%</td>
</tr>
<tr>
<td>Sepia</td>
<td>80%</td>
<td>20%</td>
<td>76.2%</td>
<td>23.8%</td>
</tr>
<tr>
<td>Smithwat</td>
<td>60%</td>
<td>40%</td>
<td>59.3%</td>
<td>40.7%</td>
</tr>
<tr>
<td>Svm</td>
<td>10%</td>
<td>90%</td>
<td>14.3%</td>
<td>85.7%</td>
</tr>
</tbody>
</table>

Adaptive mapping and manual mapping have similar distributions.
Adapting to Hardware Changes (1)

**Adaptive mapping automatically recovers part of the performance loss in the GPU from the CPU**

Using a less powerful GPU

(GTX8800 with 128 cores => GTS8800 with 96 cores)
Adapting to Hardware Changes (2)

Adaptive mapping shifts most work to the GPU

Using a less powerful CPU

(CPU with 8 cores => CPU with 2 cores)

- CPU-always
- GPU-always
- Adaptive mapping

Original result

Speedup over Serial

- Manual mapping
- Adaptive mapping
Conclusions

• A fully automated technique to map computations onto heterogeneous multicores
• Encouraging results:
  – Achieve 94% of the speedup of manual mapping
  – Nearly as good as manual mapping in energy saving
  – Adapt to changes in input size, hardware & software

☞ An interesting example of Online Autotuning
Acknowledgments

• Michael Linderman, Jamison Collins, Hong Wang
  – Sharing their Merge benchmarks
• Geoff Lowney, Robert Cohn, Kath Knobe, Minjang Kim
  – Suggestions and feedbacks
Backup
Qilin Approach: High-Level API

Qilin Programmable API:
- Provide a library of APIs (Qilin API) to express parallelizable operations
- Advantages:
  1. Alleviate compiler’s difficult job of extract parallelism from serial code
  2. Library-based approach easier to be adopted than a new language
  3. Same APIs for different languages and different target machines
A Simple Qilin Program

- A Monte Carlo (probabilistic) approach to computing the value of $\pi$

**Algorithm**

1. Generate $N$ uniformly distributed random points within the unit square.
2. Find out how many of these random points (say $M$) are within the quadrant of the unit circle.
3. Since:

   $$\frac{M}{N} = \frac{\text{Area-of-quadrant}}{\text{Area-of-unit-square}} = \frac{\pi/4}{1} = \frac{\pi}{4}$$

   $=>$ $\pi = \frac{4M}{N}$
\[ \pi \text{ Calculation: Serial Version} \]

```c
float serial_pi()
{
    const float N = 100000; // total number of random points to be generated
    float M = 0; // number of points inside quadrant
    float x, y;

    for (int i=0; i<N; i++)
    {
        // Generate a random point \((x, y)\) within the unit square
        GenerateRandomFloats(&x, 1, 0.0, 1.0);
        GenerateRandomFloats(&y, 1, 0.0, 1.0);

        // Compute the distance of the point from the origin
        const float distance_from_origin = sqrt(x*x + y*y);

        if (distance_from_origin <= 1)
        { // \((x, y)\) falls inside the quadrant
            M += 1;
        }
    }

    // Compute \(\pi\)
    const float pi = 4.0 * (M / N);

    return pi;
}
```
float qilin_pi()
{
    const float N = 100000; // total number of random points to be generated
    float x[N], y[N];

    // Generate N random points within the unit square
    GenerateRandomFloats(x, N, 0.0, 1.0);
    GenerateRandomFloats(y, N, 0.0, 1.0);

    QArrayF32 Qx = QArrayF32::Create1D(N, x);
    QArrayF32 Qy = QArrayF32::Create1D(N, y);

    // For each random point, determine if it is within the quadrant
    QArrayF32 DistFromZero = Sqrt(Qx*Qx + Qy*Qy);
    QArrayF32 WithinQuadrant = (DistFromZero <= 1.0f);

    // Sum up the total number of random points within the quadrant
    QArrayF32 S = Sum(WithinQuadrant);
    const float M = S.ToScalar();

    // Compute Pi
    const float pi = 4.0 * (M / N);

    return pi;
}
Example: Compilation of the Qilin π

Step 1: Build a DAG from Qilin API calls:

(build the DAG based on data dependency)

PE unassigned

S[0 .. N-1]

WithinQuadrant[0 .. N-1]

DistFromZero[0 .. N-1]

Sq rt

<=

1.0

array index ranges from 0 to N-1

+ * *

Qx[0 .. N-1]

Qy[0 .. N-1]
Step 2: Map Computations to PEs: (assuming Qilin decides a 50/50 work distribution)

- **CPU**
- **GPU**

```
\[ S[0 .. N-1] = \sum_{i=0}^{N/2-1} \sqrt{S[i]} \cdot \sum_{i=N/2}^{N-1} \sqrt{S[i]} + \sum_{i=0}^{N/2-1} Q_x[i] + \sum_{i=N/2}^{N-1} Q_y[i] \]
```

```
\[ \text{WithinQuadrant}[0 .. N/2-1] \]
\[ \text{DistFromZero}[0 .. N/2-1] \]
\[ \text{Qx}[0 .. N/2-1] \]
\[ \text{Qy}[0 .. N/2-1] \]

\[ \text{WithinQuadrant}[N/2 .. N-1] \]
\[ \text{DistFromZero}[N/2 .. N-1] \]
\[ \text{Qx}[N/2 .. N-1] \]
\[ \text{Qy}[N/2 .. N-1] \]
Step 3: Optimization (Operator Coalescing):

(increase the granularity of parallelization)
Step 4: Code Generation

void cpu_fun_2(float* in0, float*)
{
    void cpu_fun_0(...);
    void cpu_fun_1(...);
}

void main()
{
    T = thread_create(cpu_main);
    gpu_main();
    thread_join(T);
    cpu_fun_2(...)
}

void cpu_fun_0(float* in0, float* in2, float* out0)
{
    //Implemented with TBB
}

void cpu_fun_1(float* in0, float* in1, float* out0)
{
    //Implemented with CUDA
}

void cpu_fun_2(float* in0, float* in1, float* out0)
{
    //Implemented with TBB
}

void cpu_fun_1(float* in0, float* in1, float* out0)
{
    //Implemented with CUDA
}

void cpu_fun_0(float* in0, float* in1, float* out0)
{
    //Implemented with TBB
}

void cpu_fun_1(float* in0, float* in1, float* out0)
{
    //Implemented with CUDA
}

void cpu_fun_2(float* in0, float* in1, float* out0)
{
    //Implemented with TBB
}

void main()
{
    T = thread_create(cpu_main);
    gpu_main();
    thread_join(T);
    cpu_fun_2(...)
}

void cpu_fun_0(float* in0, float* in2, float* out0)
{
    //Implemented with TBB
}

void cpu_fun_1(float* in0, float* in1, float* out0)
{
    //Implemented with CUDA
}

void cpu_fun_2(float* in0, float* in1, float* out0)
{
    //Implemented with TBB
}

void main()
{
    T = thread_create(cpu_main);
    gpu_main();
    thread_join(T);
    cpu_fun_2(...)
}
High Adaptation Overhead in Smithwat (1)

Original serial version:

```c
void SerialSmithwat(float* score,
                    float* seqA, int startA, int lenA,
                    float* seqB, int startB, int lenB)
{
    for (int i=startA; i<lenA; i++)
        for (int j=startB; j<lenB; j++)
            ScoreOnePair(score, seqA, i, lenA, seqB, j, lenB);
}
```
High Adaptation Overhead in Smithwat (2)

```c
void MySmithwat(float* score, float* seqA, int i, int lenA,
                float* seqB, int startB, int lenB)
{
    QArray<float> q1 = QArray<float>::Create1D(...);
    QArray<float> q2 = QArray<float>::Create1D(...);

    QArrayOp mySmithwat = MakeQArrayOp(...);
    QArrayOpArgsList argList;
    argList.Insert(q1, ...); argList.Insert(q2, ...);

    QArray<BOOL> qSuccess = ApplyQArrayOp(mySmithwat, argList, ...);
    qSuccess.ToNormalArray(...);
}

void QilinSmithwat(float* score, float* seqA, int startA, int lenA,
                    float* seqB, int startB, int lenB)
{
    for (int i=startA; i<lenA; i++)
        MySmithwat(score, seqA, i, lenA, seqB, startB, lenB);
}
```
Accuracy of Qilin’s Performance Projection

Qilin’s execution-time projections in Binomial option pricing

Qilin’s performance projection is highly accurate
Impact of Training Input Size

(Note: The y-axis is in logarithmic scale)

Most of the performance benefit of Adaptive Mapping preserved when the training input size is at least 30% of the reference input size
Adapting to Software Changes

GCC doesn’t use SSE-x as well as ICC does

Adaptive mapping biases to GPU

Using a different compiler on CPU

ICC => GCC

(for both the serial and parallel cases)
Related Work

Hardware
- Kumar et al. demonstrate advantages of heterogeneous over homogeneous CMPs in terms of power and throughput.
- Similar observations from Hill and Mart.
  => Both study point out the importance of the mapping problem.

Software
- GPGPU:
  • Brook, Accelerator, Peakstream, Rapidmind, Brook+, Cuda (they are all GPU only).
- Intel’s TBB and Ct (currently CPU only).
- IBM’s OpenMP extension for Cell and Intel’s ExoCHI/Merge.
  • Use both CPU and GPU, but based on static manual mapping.
- OpenCL:
  • Doesn’t seem to have any automatic mapping technique based on the initial specification.

Autotuning
- Generating many variants of a computation kernel and benchmarking each variant on the target platform.
- Adaptive mapping can be regarded as an autotuning technique that tunes for the distribution of works on heterogeneous platforms.