CONVENTION ANNUELLE CRiP

Digital :
le moteur de l'innovation
et de la compétitivité

17 & 18 juin
Paris La Défense

IBM Power™ 8 experiments

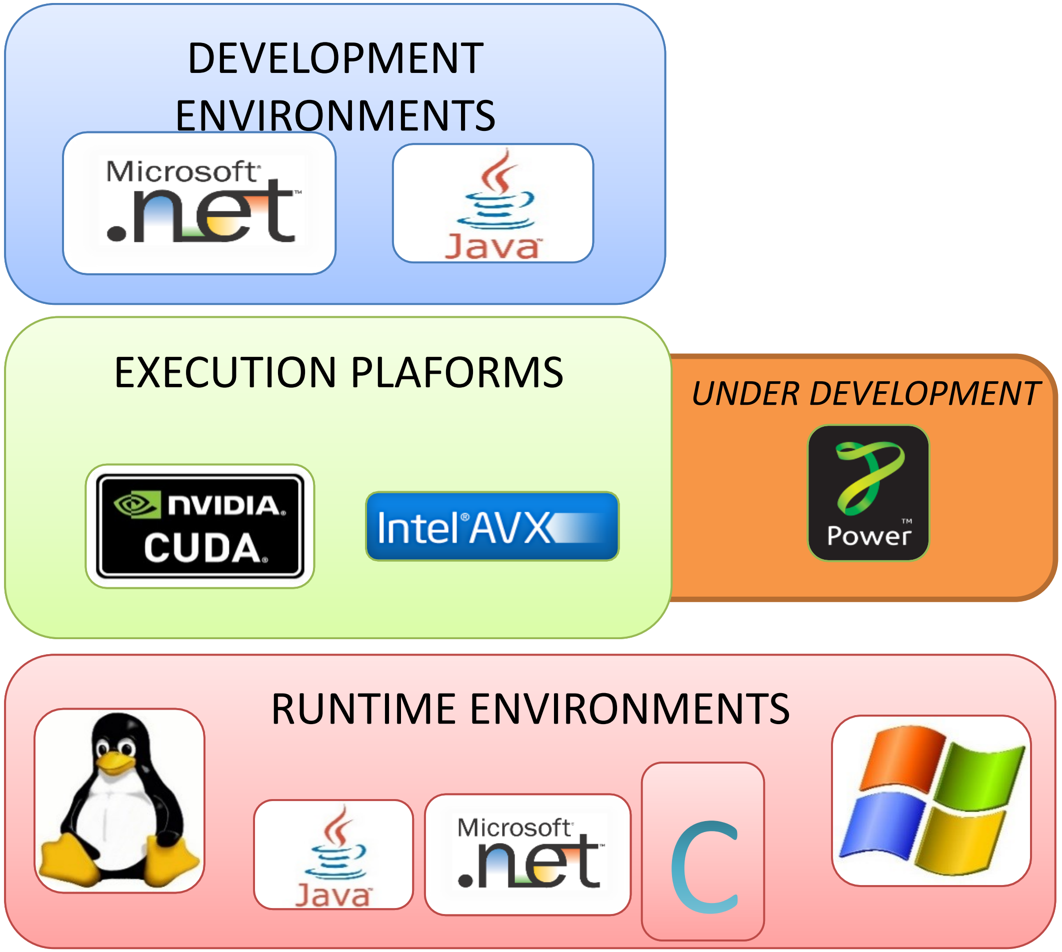
Bridging the gap between multi-core and many-core

Florent DUGUET, ALTIMESH

Motivation

- Altimesh offers a productivity tool, the Hybridizer™, to enable accelerators from dot net or java environments.
- Hybridizer™ currently supports NVIDIA GPU, Intel AVX processors, and is actively developing Xeon PHI support, AMD manycore solutions.
- Altimesh wanted to explore the capabilities of IBM Power™ 8 processor.

Altimesh Hybridizer™ Environment



DEVELOPMENT ENVIRONMENTS

EXECUTION PLAFORMS

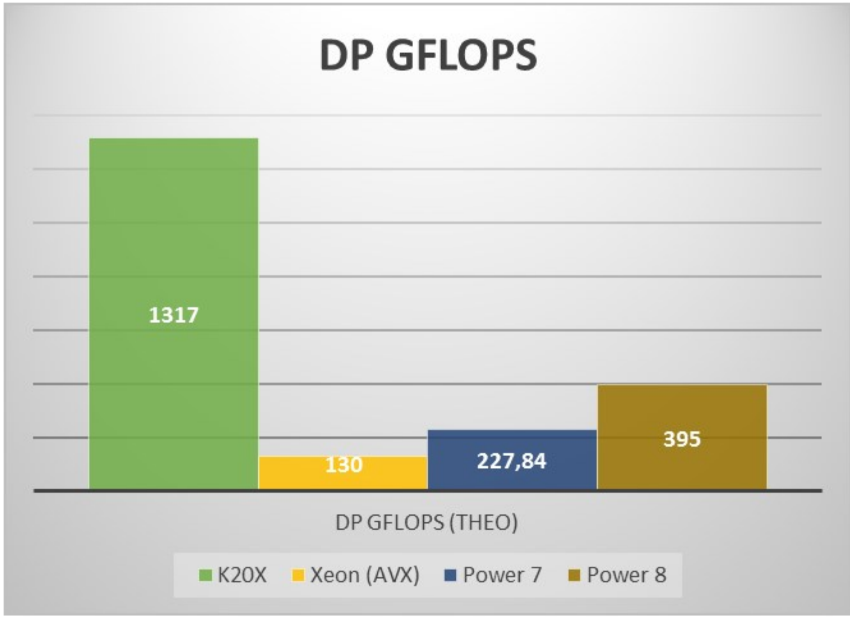
UNDER DEVELOPMENT

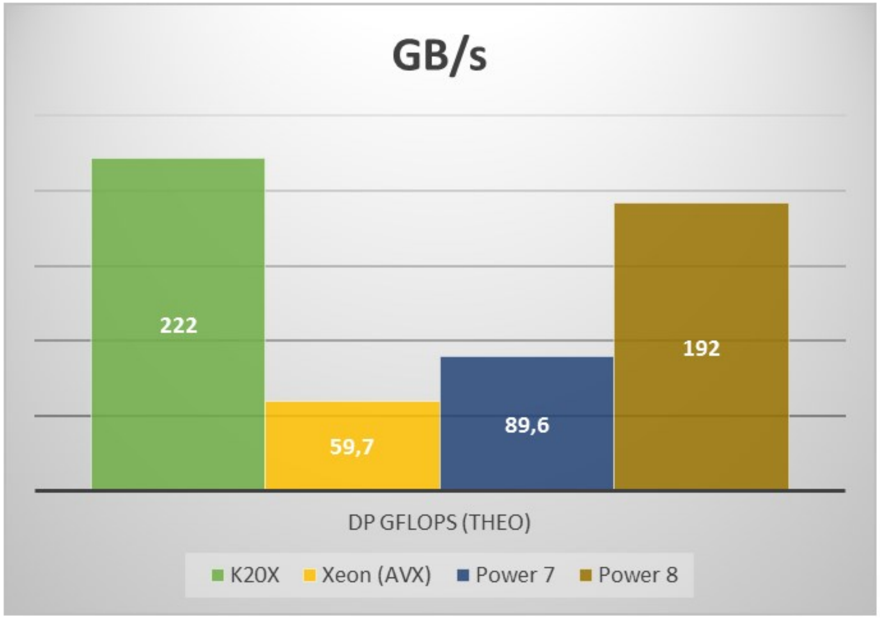
RUNTIME ENVIRONMENTS

IBM Power 8 – figures

- Bandwidth per socket (4GHz-assuming full occupancy of Centaur memory buffers)
  - 128 GB/s read from main memory
  - 64 GB/s write to main memory
- Compute per socket
  - 12 cores
  - 2 VSX per core (4 Double precision FMA per cycle, 8 SP)
  - 4.116 GHz (tested configuration)
  - 395 GFLOPs Double Precision

Peak performance at a glance





NVIDIA K20x – 14 SMX@735 MHz – 222 GB/s (ECC on) – 235 W
Intel Xeon E5-2697 – 12 cores @ 2.7 GHz – 59.7 GB/s – 130W
Power 7+ – 8 cores @ 3.56 GHz – 89.6 GB/s (51.2+38.4)
Power 8 - 12 cores @ 4.116 GHz – 192 GB/s (128+64)

Memory Bandwidth

- Memory bandwidth
  - Read - accumulate from a large table
  - Read + Write (inplace) – accumulation of one table in another
  - Read + Write (copy) – accumulation of two tables in a third

	T	R	W	Read Time	Write Time	Usage of Peak
READ	0,004759638	1	0	0,00390625	0	**82%**
R+W (inplace)	0,01162115	2	1	0,0078125	0,0078125	**67%**
R/W (copy)	0,01635323	3	1	0,01171875	0,0078125	**72%**

Reads and writes are concurrent (we can aggregate bandwidth)

Compute

[Test system is a pair of Power 8 processors @ 4.116 GHz]

- GCFLOPS: not all algorithms can benefit from fused multiply add. Counting FMA a single CFLOP, just as mul or add and counting achieved CFLOPS.

- Expm1 (Taylor expansion of exp(x)-1)

- Several implementations tested (many ways of using VSX units given compiler optimizations and inlining performances)

Test code compiled using GCC 4.8.2 : flags : -O3 -mvsx -maltivec –fopenmp -mtune=power8 -mcpu=power8 -mpower8-vector

Altimesh

http://www.altimesh.com

Compute – Power 7+

Peak	99,68	gcc 4.4.7-3			xlc 12.1		
Test configuration		**GCFLOPS**	**GFLOPS**	**Usage**	**GCFLOPS**	**GFLOPS**	**usage**
WhetStone		99.534989	99.534989	**99.85%**			
EXPM1	Naïve	34.383456	62.320014	34.49%	25.999829	47.12469	26.08%
	double4	33.347231	60.441856	33.45%	14.595009	8.052419	14.64%
	__vector double	26.866234	48.695049	26.95%			
	altidouble	27.557248	49.947511	27.65%			
	phipower<4>	27.553094	49.939983	27.64%			
	phipower<8>	50.234199	91.049486	50.40%			
	phipower<16>	24.54761	44.492543	24.63%			
	doublevect<4>	33.511902	60.740323	33.62%	9.762122	17.693846	9.79%
	doublevect<8>	10.362701	18.782396	10.40%	9.279108	16.818384	9.31%
	doublevectnoop<4>	32.438908	58.79552	32.54%	18.291425	10.091821	18.35%
	doublevectnoop<8>	50.79917	92.073496	50.96%	11.364447	20.59806	11.40%
	doublevectnoop<12>	46.942798	85.083821	47.09%	20.076538	36.388725	20.14%
	doublevectnoop<16>	24.524694	44.451008	24.60%	22.413374	40.62424	22.49%
	doublevectnoop<32>	21.12745	38.293503	21.20%	29.5969	53.644382	29.69%
	doublevectnosplit<8>	37.142535	67.320845	37.26%	37.211826	67.446435	37.33%
	doublevectnosplit<16>	14.671557	26.592196	14.72%	48.867318	88.572014	49.02%
	doublevectnosplit<32>	69.176547	125.382491	69.40%	47.136457	85.434828	47.29%
	doublevectnosplit<64>	42.096261	76.299473	42.23%	35.244512	63.880677	35.36%

Compute – Power 8

Peak	395,136	double - gcc 4.8.2			double - xlc 13.1.0			float - gcc 4.8.2		
Test configuration		**GCFLOPS**	**GFLOPS**	**usage**	**GCFLOPS**	**GFLOPS**	**usage**	**GCFLOPS**	**GFLOPS**	**usage**
WhetStone		326,86	326,86	**82,72%**				653,79	653,79	**82,73%**
EXPM1	Naïve	206,61	374,48	52,29%	33,16	60,10	8,39%	337,86	612,37	42,75%
	double4	166,98	302,64	42,26%	5,44	9,85	1,38%			
	__vector double	204,95	371,48	51,87%				342,82	623,17	43,38%
	altidouble	109,83	199,07	27,80%				218,43	395,91	27,64%
	phipower<4>	297,94	540,03	75,40%						
	phipower<8>	196,11	355,45	49,63%				574,63	1041,51	72,71%
	phipower<16>	87,65	158,87	22,18%				375,36	680,34	47,50%
	doublevect<4>	111,51	202,11	28,22%	5,79	10,50	1,47%	362,32	656,70	45,85%
	doublevect<8>	140,09	253,92	35,45%	6,48	11,75	1,64%	112,19	203,35	14,20%
	doublevectnoop<4>	161,30	292,36	40,82%	5,46	9,90	1,38%	220,33	399,35	27,88%
	doublevectnoop<8>	184,47	334,36	46,69%	10,78	19,53	2,73%	195,27	353,97	24,71%
	doublevectnoop<12>	147,83	267,94	37,41%	11,80	21,39	2,99%	306,59	555,69	38,80%
	doublevectnoop<16>	89,20	161,68	22,57%	13,86	25,13	3,51%	211,32	383,02	26,74%
	doublevectnoop<32>	92,01	166,76	23,29%	11,70	21,22	2,96%	216,08	391,64	27,34%
	doublevectnosplit<8>	157,85	286,09	39,95%	132,31	239,81	33,48%	223,17	404,49	28,24%
	doublevectnosplit<16>	91,73	166,25	23,21%	167,84	304,22	42,48%	309,79	561,50	39,20%
	doublevectnosplit<32>	82,49	149,51	20,88%	136,02	246,54	34,42%	182,88	331,47	23,14%
	doublevectnosplit<64>	68,35	123,88	17,30%	227,83	412,95	57,66%	168,14	304,75	21,28%

Compute

[Test system is a pair of Power 8 processors @ 4.116 GHz]

- GCFLOPS: not all algorithms can benefit from fused multiply add. Counting FMA a single CFLOP, just as mul or add and counting achieved CFLOPS.

- Expm1 (Taylor expansion of exp(x)-1)

Test	Implem	Double Precision	Usage	Single Precision	Usage
Whetstone	Optimized	326.86	82.72 %	653.79	82.73 %
Expm1	Naïve	206.61	52.29 %	337.86	42.75 %
Expm1	Optimized	297.94	75.40 %	574.63	72.71 %

Test code compiled using GCC 4.8.2 : flags : -O3 -mvsx -maltivec –fopenmp -mtune=power8 -mcpu=power8 -mpower8-vector

Use Case Benchmark

- Fixed cash flows pricer – accumulate discounts of cash flows with linear interpolation on the interest rate

$$\pi = \sum_{cash\ flows} N * e^{-T * r\,(T)}$$

$$r(T) = \frac{T - T_-}{T_+ - T_-} * r_+ + \frac{T_+ - T}{T_+ - T_-} * r_-$$

- Implementations : Java, Default (C++), optimized with FMA, optimized without FMA. All implementations have same algorithmic optimizations (precalculated lookups and interpolations).

Default implementation of exp seems to be the biggest performance blocker

Use Case Benchmark

- Fixed cash flows pricer – accumulate discounts of cash flows with linear interpolation on the interest rate

Implementation	Double precision (Million CF/s)	Ratio with Java	Single precision (Million CF/s)	Ratio with Java
Java	673	1.0	744	1.0
Default (C++)	686	1.0	N/A	
Optimized no-FMA	713	10.7	14,365	19.3
Optimized FMA	10,290	15.3	17,740	23.8

Default implementation of exp seems to be the biggest performance blocker

Comparing to other platforms

Million Cashflows	IBM	Intel	NVIDIA
Discountings per second	Power 8	i7 - 4771	K20c
Java - Float (1)	744	N/A	N/A
Java - Double	673	N/A	N/A
C# - Float (1)	N/A	325	N/A
C# - Double	N/A	359	N/A
Optimized - Float	17740	1339	23628
Optimized - Double	10290	1309	9426

(1): in Java or DotNet APIs, single precision operations are not exposed.

Wrap up

- Accelerator-grade performance (memory and compute)
- CPU-grade flexibility
- Large caches
- No vectorization does not totally sacrifice performances (1/2 compared to 1/4 for Intel CPU)
- Bigger nodes to reduce the costs of sysadmin