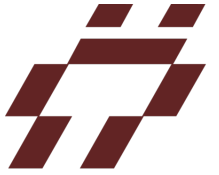




Возможности использования процессоров Иртыш для научных вычислений: Ускорение вычислений

Владислав Щапов

Кто я?



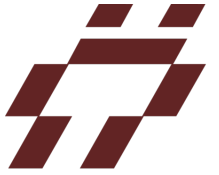
- К.Т.Н., доцент
- Разработчик в Трамплин Электроникс
- Доцент Пермского национального исследовательского политехнического университета
- Контрибьютор в OpenSource
 - В том числе реализовал оптимизации кода под LoongArch
- Email: vladislav@shchapov.ru
- URL: <https://www.shchapov.ru>





В предыдущих сериях

Процессор Иртыш



- LoongArch
- 64 бит
- RISC архитектура
- Weak memory ordering
- Производительность на уровне





Программная экосистема

Средства разработки

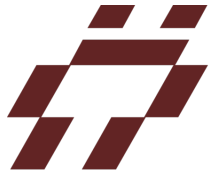


- GCC 14+ (для LSX/LASX лучше 16+)
- LLVM (Clang) 18+ (для LSX/LASX лучше 22+)
- Rust 1.71+
- Go 1.19+

- Порты от Loongson:
 - [OpenJDK](#) (8, 11, 17, 21, 25)
 - [.NET](#)

- Рекомендация: Использовать последние версии компиляторов.
- Стандартные средства разработки: Кроссплатформенный код переносится без проблем.

Оптимизированные библиотеки

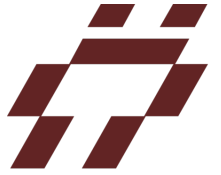


Название	Минимальная версия	LSX	LASX
fftw3	3.3.11	✓	✓
OpenBLAS	0.3.18+ (LSX, LASX)	✓	✓
Eigen	5.0.0+ (LSX)	✓	
highway	1.4.0+ (LSX, LASX)	✓	✓
numpy	2.3.0	✓	
OpenCV	5	✓	✓
libpng	1.6.41	✓	
simdjson		✓	✓
zlib-ng	2.3	✓	✓
PCRE2	10.43	✓	



Способы ускорения вычислений

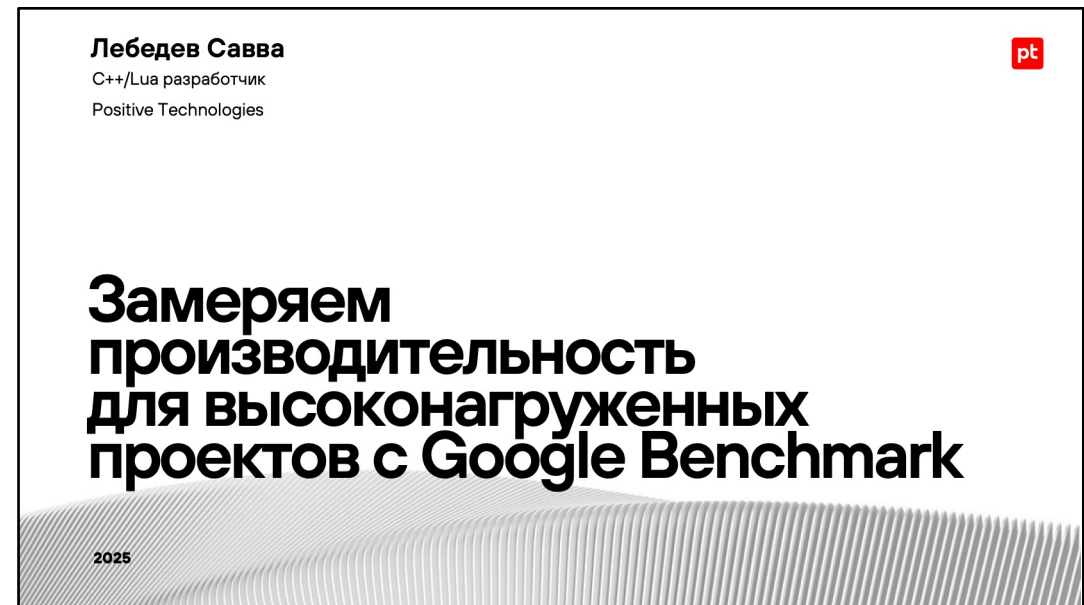
Измерение производительности



- C++ Russia 2025: Савва Лебедев «Замеряем производительность для высоконагруженных проектов с Google Benchmark»
- Видео: <https://youtu.be/zP8PB7Q3z6o>



- Слайды: <https://jrg.su/9dRNNa>



Привычный мир: SSE / AVX / FMA



SSE (регистр 128 бит)

- Типы данных:
 - float, double
 - char, short, int, long, long long
- Все основные операции:
 - Математические
 - Логические
 - Битовые
 - Преобразования типов
 - ...

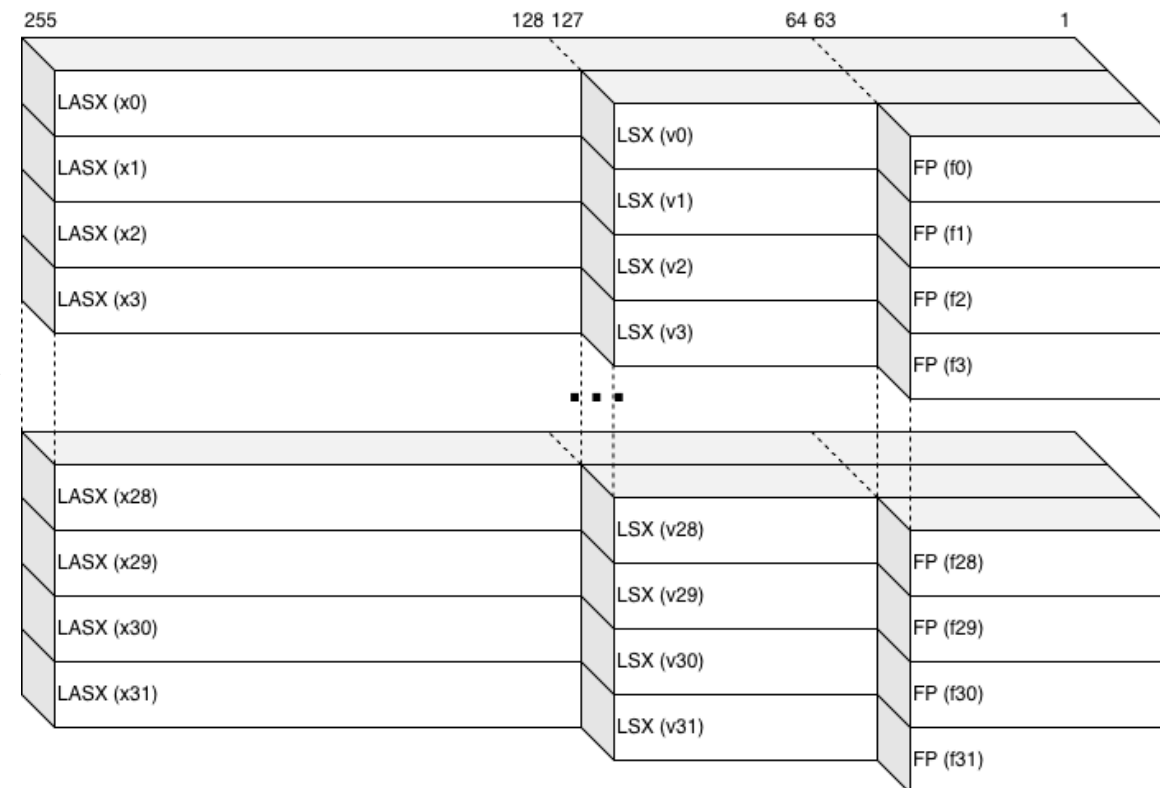
AVX / FMA (регистр 256 бит)

- Типы данных:
 - float, double (AVX)
 - char, short, int, long, long long
- AVX: «два по 128 бит»:
 - float, double
- AVX2: «int + 256 бит + mask»
 - int*
 - permutevar8x32
 - Gather
 - mask{load,store}
- FMA: $a = a \pm b \cdot c$

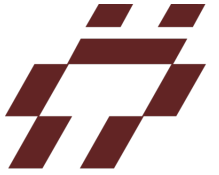
LoongArch: LSX и LASX



- Типы данных:
 - float, double, char, short, int, long, long long
- LSX: 32 регистра по 128 бит
- LASX: 32 регистра по 256 бит
- Младшие биты регистров FP/LSX/LASX общие:
 - LSX регистр – это младшая половина LASX регистра
 - FP регистр – это младшая половина LSX регистра
- Операции:
 - Базовые математические, логические, битовые, ...
 - FMA
- Особенность LSX:
 - Нет гарантии атомарности чтения из памяти 128-битных выровненных данных.



Путь 1: Авто векторизация



```
1 #include <stddef.h>
2
3 void saxpy(
4     const size_t n,
5     const float a,
6     const float *restrict x,
7     float *restrict y
8 ) {
9     for (size_t i = 0; i < n; ++i)
10         y[i] = a*x[i]+y[i];
11 }
12
```

```
16 .L4:
17     vmulps ymm0, ymm1, YMMWORD PTR [r8+rax]
18     vaddps ymm0, ymm0, YMMWORD PTR [rdx+rax]
19     vmovups YMMWORD PTR [rdx+rax], ymm0
20     add rax, 32
21     cmp rax, rcx
22     jne .L4
23     lea rax, [0+rsi*8]
24     cmp rdi, rax
25     je .L11
26 .L3:
27     mov rcx, rdi
28     sub rcx, rax
29     lea rsi, [rcx-1]
30     cmp rsi, 2
31     jbe .L5
32     vmulps xmm2, xmm2, XMMWORD PTR [r8+rax*4]
33     mov rsi, rcx
34     vaddps xmm2, xmm2, XMMWORD PTR [rdx+rax*4]
35     and rsi, -4
36     and ecx, 3
```

```
1 saxpy:
2     beqz $r4,.L1
3     addi.d $r12,$r4,-1
4     addi.w $r13,$r0,14 # 0xe
5     bleu $r12,$r13,.L7
6     srli.d $r14,$r4,4
7     xvreplve0.w $xr3,$xr0
8     slli.d $r15,$r14,6
9     or $r13,$r5,$r0
10    or $r12,$r6,$r0
11    add.d $r15,$r5,$r15
12 .L4:
13    xvld $xr5,$r12,32
14    xvld $xr4,$r12,0
15    xvld $xr2,$r13,32
16    xvld $xr1,$r13,0
17    xvfmadd.s $xr2,$xr2,$xr3,$xr5
18    xvfmadd.s $xr1,$xr1,$xr3,$xr4
19    xvst $xr2,$r12,32
20    xvst $xr1,$r12,0
21    addi.d $r13,$r13,64
```

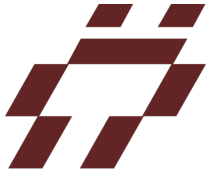
<https://godbolt.org/z/z95GGxaWW>



```
<source>:9:26: optimized: loop vectorized using 32 byte vectors
and unroll factor 8
<source>:9:26: optimized: epilogue loop vectorized using 16 byte
vectors and unroll factor 4
<source>:9:26: optimized: loop with 2 iterations completely
unrolled (header execution count 32134393)
<source>:10:9: optimized: loop turned into non-loop; it never
loops
Compiler returned: 0
```

```
<source>:9:26: optimized: loop vectorized using 32 byte vectors
and unroll factor 16
<source>:9:26: optimized: epilogue loop vectorized using 16
byte vectors and unroll factor 4
<source>:9:26: optimized: loop with 2 iterations completely
unrolled (header execution count 34171080)
<source>:10:9: optimized: loop with 2 iterations completely
unrolled (header execution count 39682545)
Compiler returned: 0
```

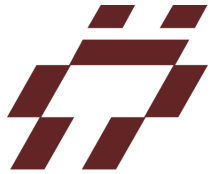
Путь 2: Библиотека SIMD Everywhere



- SIMD Everywhere – это библиотека, предоставляющая набор заголовочных файлов, которые содержат быстрые и переносимые реализации SIMD-функций на архитектуры, которые их изначально не поддерживают, например, вызов функций SSE на LoongArch.
- Лицензия: MIT
- URL: <https://github.com/simd-everywhere/simde/>



Путь 2: Библиотека SIMD Everywhere



C source #1

Save/Load + Add new... Vim C

```
1 #include <stddef.h>
2 #ifdef __loongarch__
3 # define SIMDE_ENABLE_NATIVE_ALIASES
4 # include <simd/x86/fma.h>
5 #else
6 # include <immintrin.h>
7 #endif
8
9 void saxpy(
10     const size_t n,
11     const float a,
12     const float *restrict x,
13     float *restrict y
14 ) {
15     __m256 a_v = __m256_set1_ps(a);
16     size_t i = 0;
17     for (; i < (n & ~((size_t)7)); i += 8) {
18         __m256 x_v = __m256_loadu_ps(&x[i]);
19         __m256 y_v = __m256_loadu_ps(&y[i]);
20         __m256 t = __m256_fmadd_ps(a_v, x_v, y_v);
21         __m256_storeu_ps(&y[i], t);
22     }
23     for (; i < n; ++i)
24         y[i] = a*x[i]+y[i];
25 }
```

<https://godbolt.org/z/qaTP4xoKo>



x86-64 gcc 16.2 (Editor #1)

x86-64 gcc 16.2 -O3 -mfma -fno-tree-vectorize

```
1 "saxpy":
2     vmovaps xmm1, xmm0
3     mov     rcx, rdx
4     cmp     rdi, 7
5     jbe     .L6
6     mov     rax, rdi
7     vshufps xmm2, xmm0, xmm0, 0
8     xor     edx, edx
9     vinsertf128 ymm2, ymm2, xmm2, 1
10    and     rax, -8
11    .L3:
12        vmovups ymm0, YMMWORD PTR [rsi+rdx*4]
13        vfmadd213ps ymm0, ymm2, YMMWORD PTR [rcx+rdx*4], ymm0
14        vmovups YMMWORD PTR [rcx+rdx*4], ymm0
15        add     rdx, 8
16        cmp     rdx, rax
17        jb     .L3
18        vzeroupper
19    .L2:
20        cmp     rax, rdi
21        jnb     .L9
22    .L5:
23        vmovss xmm0, DWORD PTR [rsi+rax*4]
24        vfmadd213ss xmm0, xmm1, DWORD PTR [rcx+rax*4], xmm0
25        vmovss DWORD PTR [rcx+rax*4], xmm0
26        add     rax, 1
27        cmp     rdi, rax
28        jne     .L5
29    .L9:
```

Output (0/0) [already open] x86-64 gcc 16.2 i - 469ms (20964B) ~1400 lines filtered

Output of x86-64 gcc 16.2 (Compiler #1)

Wrap lines Select all

Compiler returned: 0

loongarch64 gcc 16.1.0 (Editor #1)

loongarch64 gcc 16.1.0 -O3 -mlasx -fno-tree-vectorize

```
1 saxpy:
2     addi.d $r3,$r3,-16
3     addi.w $r13,$r0,7 # 0x7
4     fst.s $f0,$r3,12
5     or     $r12,$r0,$r0
6     bleu  $r4,$r13,.L2
7     movfr2gr.s $r14,$f0
8     or     $r12,$r4,$r0
9     xvreplgr2vr.w $xr3,$r14
10    bstrins.d $r12,$r0,2,0
11    or     $r15,$r5,$r0
12    or     $r13,$r6,$r0
13    or     $r14,$r0,$r0
14    .L3:
15        xvld    $xr1,$r15,0
16        xvld    $xr2,$r13,0
17        addi.d $r14,$r14,8
18        xvfmadd.s $xr1,$xr3,$xr1,$xr2
19        xvst    $xr1,$r13,0
20        addi.d $r15,$r15,32
21        addi.d $r13,$r13,32
22        bltu   $r14,$r12,.L3
23    .L2:
24        bleu  $r4,$r12,.L1
25        als1.d $r13,$r12,$r5,2
26        als1.d $r6,$r12,$r6,2
27        als1.d $r4,$r4,$r5,2
28    .L5:
29        fld.s  $f2,$r6,0
```

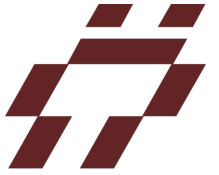
Output (0/0) [already open] loongarch64 gcc 16.1.0 i - 455ms (21371B) ~1353

Output of loongarch64 gcc 16.1.0 (Compiler #2)

Wrap lines Select all

Compiler returned: 0

Путь 3: Ручная векторизация



- Если есть базовый код для SSE/AVX:
 - Замена интринсиков SSE/AVX на аналогичные по функциональности интринсики LSX/LASX.
 - Разработка функциональных аналогов для интринсиков, отсутствующих в LSX/LASX.
 - Анализ производительности.
- Если нет SSE/AVX версии:
 - Векторизация с учетом списка инструкций LoongArch.
 - Анализ производительности.

Путь 3: Ручная векторизация

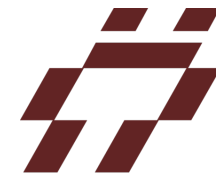


SSE	LSX	AVX	LASX
<code>_mm_abs_epi16</code>	<code>__lsx_vsigncov_h</code>	<code>_mm256_abs_epi16</code>	<code>__lasx_xvsigncov_h</code>
<code>_mm_abs_epi32</code>	<code>__lsx_vsigncov_w</code>	<code>_mm256_abs_epi32</code>	<code>__lasx_xvsigncov_w</code>
<code>_mm_abs_epi8</code>	<code>__lsx_vsigncov_b</code>	<code>_mm256_abs_epi8</code>	<code>__lasx_xvsigncov_b</code>
<code>_mm_add_epi16</code>	<code>__lsx_vadd_h</code>	<code>_mm256_add_epi16</code>	<code>__lasx_xvadd_h</code>
<code>_mm_add_epi32</code>	<code>__lsx_vadd_w</code>	<code>_mm256_add_epi32</code>	<code>__lasx_xvadd_w</code>
<code>_mm_add_epi64</code>	<code>__lsx_vadd_d</code>	<code>_mm256_add_epi64</code>	<code>__lasx_xvadd_d</code>
<code>_mm_add_epi8</code>	<code>__lsx_vadd_b</code>	<code>_mm256_add_epi8</code>	<code>__lasx_xvadd_b</code>
<code>_mm_add_pd</code>	<code>__lsx_vfadd_d</code>	<code>_mm256_add_pd</code>	<code>__lasx_xvfadd_d</code>
<code>_mm_add_ps</code>	<code>__lsx_vfadd_s</code>	<code>_mm256_add_ps</code>	<code>__lasx_xvfadd_s</code>
<code>_mm_adds_epi16</code>	<code>__lsx_vsadd_h</code>	<code>_mm256_adds_epi16</code>	<code>__lasx_xvsadd_h</code>
<code>_mm_adds_epi8</code>	<code>__lsx_vsadd_b</code>	<code>_mm256_adds_epi8</code>	<code>__lasx_xvsadd_b</code>

https://jia.je/unofficial-loongarch-intrinsics-guide/migrating_sse/

https://jia.je/unofficial-loongarch-intrinsics-guide/migrating_avx/

Векторизация zlib-ng



- **zlib-ng - zlib data compression library for the next generation systems**
- **Оптимизации для архитектур:**
 - **ARM, LoongArch64, Power, RISC-V, S390x, x86**
- **Оптимизация для LoongArch64:**
 - **Ручная векторизация на основе SSE и AVX кода.**

Векторизация zlib-ng



x86_64 AVX2

```
static inline void chunkmemset_2(uint8_t *from, chunk_t *chunk) {
    *chunk = _mm256_set1_epi16(zng_memread_2(from));
}

static inline void chunkmemset_4(uint8_t *from, chunk_t *chunk) {
    *chunk = _mm256_set1_epi32(zng_memread_4(from));
}

static inline void chunkmemset_8(uint8_t *from, chunk_t *chunk) {
    *chunk = _mm256_set1_epi64x(zng_memread_8(from));
}

static inline void chunkmemset_16(uint8_t *from, chunk_t *chunk) {
    /* See explanation in chunkset_avx512.c */
    #if defined(_MSC_VER) && _MSC_VER < 1920
        halfchunk_t half = _mm_loadu_si128((__m128i*)from);
        *chunk = _mm256_inserti128_si256(_mm256_castsi128_si256(half), half, 1);
    #else
        *chunk = _mm256_broadcastsi128_si256(_mm_loadu_si128((__m128i*)from));
    #endif
}
```

LoongArch64 LASX

```
static inline void chunkmemset_2(uint8_t *from, chunk_t *chunk) {
    *chunk = __lasx_xvreplgr2vr_h(zng_memread_2(from));
}

static inline void chunkmemset_4(uint8_t *from, chunk_t *chunk) {
    *chunk = __lasx_xvreplgr2vr_w(zng_memread_4(from));
}

static inline void chunkmemset_8(uint8_t *from, chunk_t *chunk) {
    *chunk = __lasx_xvreplgr2vr_d(zng_memread_8(from));
}

static inline void chunkmemset_16(uint8_t *from, chunk_t *chunk) {
    *chunk = lasx_broadcast_128(__lsx_vld(from, 0));

    #ifndef __loongarch_asx_sx_conv
    static inline __m256i __lasx_concat_128(__m128i lo, __m128i hi) {
        __m256i dest;
        __asm__ volatile ("xvperm.q %u0,%u2,0x02\n" : "=f"(dest) : "0"(lo), "f"(hi));
        return dest;
    }
    #endif

    static inline __m256i lasx_broadcast_128(__m128i in) {
        return __lasx_concat_128(in, in);
    }
}
```

Векторизация zlib-ng



x86_64 AVX2

```
__m256i vs1_sad = _mm256_sad_epu8(vbuf, zero);  
__m256i vs1_sad2 = _mm256_sad_epu8(vbuf_0, zero);
```

```
FOR j := 0 to 31  
  i := j*8  
  tmp[i+7:i] := ABS(a[i+7:i] - b[i+7:i])  
ENDFOR  
FOR j := 0 to 3  
  i := j*64  
  dst[i+15:i] := tmp[i+7:i] + tmp[i+15:i+8] + tmp[i+23:i+16] + tmp[i+31:i+24] +  
                 tmp[i+39:i+32] + tmp[i+47:i+40] + tmp[i+55:i+48] + tmp[i+63:i+56]  
  dst[i+63:i+16] := 0  
ENDFOR
```

LoongArch64 LASX

```
__m256i vs1_sad = lasx_sad_bu(vbuf, zero);  
__m256i vs1_sad2 = lasx_sad_bu(vbuf_0, zero);
```

```
static inline __m256i lasx_sad_bu(__m256i a, __m256i b) {  
  __m256i tmp = __lasx_xvabsd_bu(a, b);  
  tmp = __lasx_xvhaddw_hu_bu(tmp, tmp);  
  tmp = __lasx_xvhaddw_wu_hu(tmp, tmp);  
  return __lasx_xvhaddw_du_wu(tmp, tmp);  
}
```

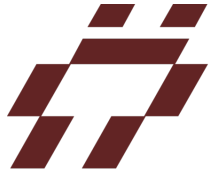
Эффективность векторизации zlib-ng

- x86_64 (AVX2):
 - Сжатие: +4,56%
 - Распаковка: +38,66%
- ARM64:
 - Сжатие: +1,13%
 - Распаковка: +24,06%
- LoongArch64 (LASX):
 - Сжатие: +3,34%
 - Распаковка: +39,22%
- Тест: [deflatebench](#).



А если процессоров мало?

AMD ROCm – расчеты на GPU



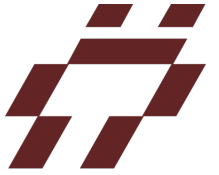
- **Установка в AOSC OS**

```
sudo oma install rocm-base rocm-core
```

- **Сборка ROCm Examples**

```
git clone -b release/therock-7.14 https://github.com/ROCm/rocm-  
examples.git  
cd rocm-examples  
export ROCM_HOME=/usr/lib/rocm  
export ROCM_PATH=/usr/lib/rocm  
export HIP_PATH=/usr/lib/rocm  
export PATH=/usr/lib/rocm/bin:/usr/lib/rocm/llvm/bin:$PATH  
export LD_LIBRARY_PATH=/usr/lib/rocm/lib:$LD_LIBRARY_PATH  
sed -i -e 's/x86_64-unknown/loongarch64-unknown/g' ./HIP-  
Basic/llvm_ir_to_executable/CMakeLists.txt  
sed -i -e 's/x86_64-unknown/loongarch64-unknown/g' ./HIP-  
Basic/assembly_to_executable/CMakeLists.txt  
cmake -S . -B build  
cmake --build build --verbose
```

AMD ROCm



- Получение информации о карточке:

```
$ ./build/bin/HIP-Basic/hip_device_query
```

```
Hip target platform:      AMD
```

```
Compiler:                 HIP-Clang
```

```
-----  
Device ID                 0  
-----
```

```
Name:                    AMD Radeon RX 6600
```

```
totalGlobalMem:          7.98 GiB
```

```
sharedMemPerBlock:       64 KiB
```

```
regsPerBlock:            131072
```

```
warpSize:                32
```

```
maxThreadsPerBlock:      1024
```

```
maxThreadsDim:           (1024, 1024, 1024)
```

```
maxGridSize:             (2147483647, 65535, 65535)
```

```
clockRate:               2750 Mhz
```

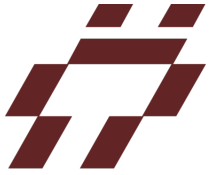
```
gcnArchName:             gfx1032
```

```
peer(s) :                None
```

```
memInfo.total:           7.98 GiB
```

```
memInfo.free:            7.95 GiB (100%)
```

AMD ROCm



- **Запуск примеров на видеокарте:**

```
$ ./build/bin/Applications/applications_histogram
```

```
Launching 'histogram256_block' with 8 blocks of size 128
```

```
Kernel took 1.51308 milliseconds.
```

```
Validation passed.
```

```
$ ./build/bin/HIP-Basic/hip_hello_world
```

```
Hello world from host!
```

```
Hello world from device or host!
```

```
Hello world from device or host!
```

```
Hello world from device or host!
```

```
Hello world from device or host!
```

```
Hello world from device or host!
```

```
Hello world from device kernel block 1 thread 0!
```

```
Hello world from device kernel block 1 thread 1!
```

```
Hello world from device kernel block 0 thread 0!
```

```
Hello world from device kernel block 0 thread 1!
```

```
$ ./build/bin/HIP-Basic/hip_matrix_multiplication
```

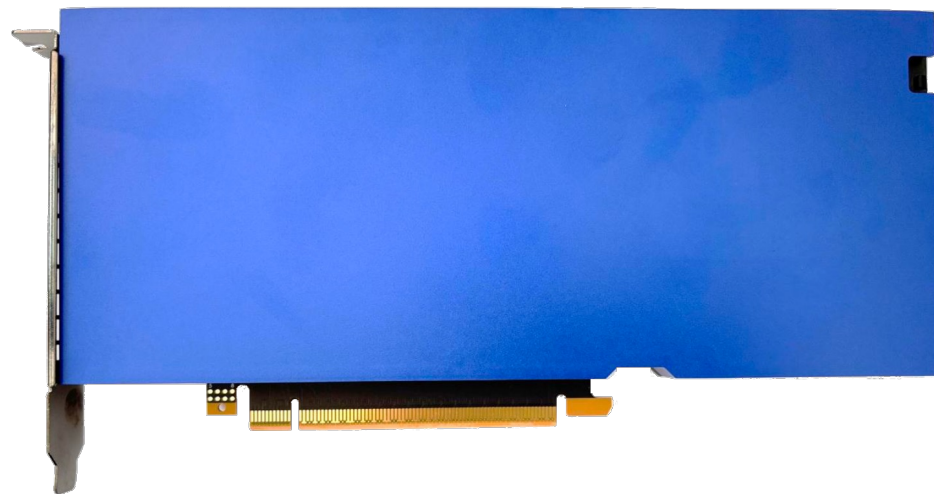
```
Matrix multiplication: [2048x1024] * [1024x1024], block size: 16x16
```

```
Validation passed.
```

Elemem – ускоритель AI



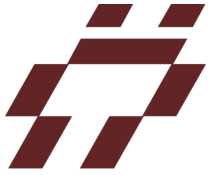
- Типы данных: UINT4, INT4, UINT8, INT8, UINT32, INT32, FP16, BF16, FP32, TF32
- Фреймворки для ИИ: PyTorch, ONNX, vLLM, TensorFlow, PaddlePaddle, DeepSpeed и др.
- Реализует CUDA 11 совместимый язык.
- Поддержка работы на процессорах LoongArch64.





Как начать погружение в Иртыш

Кросскомпилятор и эмулятор



- Доступные версии кросскомпиляторов gcc
 - Ubuntu 24.04:
 - gcc-14
 - Ubuntu 26.04:
 - gcc-15
 - gcc-16
- Эмулятор qemu
 - Можно тестировать сборки для LoongArch на x86_64, aarch64/arm64

Кросскомпиляция в Ubuntu



- **Запустим Docker контейнер**

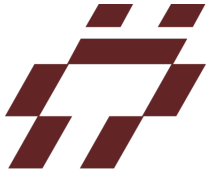
```
docker run --rm -it --net=host ubuntu:26.04 bash
```

- **Установим зависимости**

```
apt-get update
```

```
apt-get install qemu-user file gcc-16-loongarch64-  
linux-gnu g++-16-loongarch64-linux-gnu libc6-dev-  
loong64-cross
```

Кросскомпиляция в Ubuntu



```
root@docker-desktop:/# cat << 'EOF' > /test.c
```

```
#include <stdio.h>
```

```
int main() {
```

```
#ifdef __loongarch__
```

```
    printf("This is LoongArch! :)\n");
```

```
#else
```

```
    printf("This is NOT LoongArch! :(\n");
```

```
#endif
```

```
    return 0;
```

```
}
```

```
EOF
```

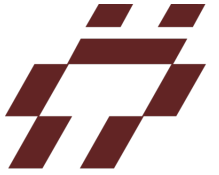
```
root@docker-desktop:/# loongarch64-linux-gnu-gcc-16 -O3 /test.c && file a.out
```

```
a.out: ELF 64-bit LSB executable, LoongArch, version 1 (SYSV), dynamically linked,  
interpreter /lib64/ld-linux-loongarch-lp64d.so.1,  
BuildID[sha1]=ee785fa597e7b75cef9ce952987be4e498631b69, for GNU/Linux 5.19.0, not stripped
```

```
root@docker-desktop:/# qemu-loongarch64 -cpu la464-loongarch-cpu -L /usr/loongarch64-  
linux-gnu/ a.out
```

```
This is LoongArch! :)
```

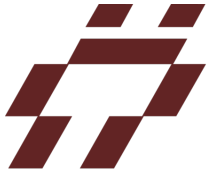
Документация



- LoongArch Documentation
- <https://loongson.github.io/LoongArch-Documentation/README-EN.html>
- Unofficial LoongArch Intrinsic Guide
- <https://jia.je/unofficial-loongarch-intrinsics-guide/>
- Software Development and Build Convention for LoongArch™ Architectures
- <https://github.com/loongson/la-softdev-convention>



Заключение



- Иртыш:
- Современная архитектура LoongArch64
- Хорошая производительность
- Стандартные инструменты разработки
- Поддержка в библиотеках
- Совместимость с ускорителями
- Используйте наши процессоры для своих расчетов!



Спасибо за внимание!

Вместе создаем технологическое будущее России!

Контакты

 partnership@tramplin.group для партнеров

 edu@tramplin.group для вузов

 tramplin.group